Chapter 1 - Introduction to PowerShell **:** introduces you to editors, the help system, command naming, and more.

Chapter 2 - Modules and Snap-Ins **:** explores finding, installing, and using modules in PowerShell. Snap-ins are not part of PowerShell 7 but are briefly explored as a legacy feature of PowerShell 5.

Chapter 3 - Working with Objects in PowerShell **:** looks at the concept of objects in PowerShell and the generic commands available for selecting, filtering, and manipulating values.

Chapter 4 - Operators **:** explores the large variety of operators available in PowerShell.

Chapter 5 - Variables, Arrays, and Hashtables *:* The chapter explores the use of variables, as well as the capabilities of collections.

Chapter 6 - Conditional Statements and Loops : are the tools used to make decisions in scripts in PowerShell. This chapter explores keywords like `if` `if` , and the different loop styles available.

Chapter 7 - Working with .NET **:** is used to dive into .NET, which was used to create the PowerShell language and is available within PowerShell.

Chapter 8 - Strings, Numbers, and Dates **:** are a vital part of any scripting language, and PowerShell is no exception. This chapter explores the different techniques available for working with such values.

Chapter 9 - Regular Expressions **:** are an incredibly useful inclusion in PowerShell. You can use regular expressions to make short work of string parsing tasks. The chapter ends by walking through several practical parsing examples.

Chapter 10 - File, Folders, and the Registry **:** explores the use of providers in PowerShell, most used to access the file system and, in Windows, the registry.

Chapter 11 - Windows Management Instrumentation **:** explores WMI in PowerShell, a significant part of the Windows operating system since Windows NT.

Chapter 12 - Working with HTML, XML, and JSON **:** looks at the PowerShell commands and .NET types that you can use to work with these different text-based formats.

Chapter 13 - Web Requests and Web Services **:** explores basic web requests before diving into using PowerShell to work with REST APIs, using the API for GitHub as an example. Support for SOAP in PowerShell 7 is less complete than in PowerShell 5.1. SOAP is explored by way of a web service project via Visual Studio.

Chapter 14 - Remoting and Remote Management **:** examines the configuration and use of PowerShell Remoting.

Chapter 15 - Asynchronous Processing **:** dives into the realm of background jobs in PowerShell before exploring .NET events in PowerShell. The chapter ends with a look at runspaces and runspace pools.

Chapter 16 - GUI **:** shows you how to implement responsive user interfaces in PowerShell.

Chapter 17 - Scripts, Functions, and Script Blocks **:** explores the building blocks of larger scripts and modules. The chapter looks at how to define parameters, work in a pipeline, and manage output.

Chapter 18 - Parameters, Validation, and Dynamic Parameters **:** looks at the many options available for defining parameters and validating input in PowerShell.

Chapter 19 - Classes and Enumerations **:** shows off the capabilities of the `class` and `enum` keywords, which were introduced with PowerShell 5. The chapter includes an exploration of class inheritance and implementing .NET interfaces. This chapter includes a brief look at writing class-based DSC resources.

Chapter 20 - Building Modules **:** explores the key concepts of creating a module in PowerShell using PowerShell code. The chapter shows off some of the common approaches available to module authors.

Chapter 21 - Testing **:** is used to explore static analysis using `PSScriptAnalyzer` as well as acceptance and unit testing using the Pester framework.

Chapter 22 - Error Handling **:** looks at the complex topic of handling errors in PowerShell, including an exploration of both terminating and non-terminating errors.

Chapter 23 - Debugging and Troubleshooting **:** uses the built-in debugger in PowerShell and Visual Studio to delve into some of the common problems encountered when debugging scripts.

EXPERT INSIGHT

Mastering PowerShell Scripting

Automate and manage your environment using PowerShell 7.1

Fourth Edition



Chris Dent

Packt

1

Introduction to PowerShell

PowerShell is a shell scripting language from Microsoft originally released for Windows in 2006. PowerShell was written to appeal to systems administrators and presents a great deal of its functionality as commands such as `Get-ChildItem`, `Get-Content`, `New-Item`, and so on. Microsoft Exchange was one of the first systems to embrace PowerShell with the release of Exchange 2007.

Windows PowerShell includes the original version through to 5.1, which is the final release of the Windows-specific shell. Windows PowerShell versions are based on .NET Framework.

In 2018, the first version of PowerShell Core was released with PowerShell 6. The move from .NET Framework to .NET Core allows the current versions of PowerShell to run on Linux and macOS, as well as Windows.

The use of .NET Core 2 in PowerShell 6 introduced a challenge for PowerShell users as it did not include everything that was in.NET Framework. For example, it was not possible to use scripts that created Windows Forms user interfaces in PowerShell 6.

In PowerShell 7, .NET Core 3.1 is used. This update reintroduced support for many of the missing features, including support for Windows Forms.

The other significant difference between Windows PowerShell and the current versions of PowerShell is that PowerShell is now open source. The project is on GitHub and is open to public contributors: https://github.com/powershell/powershell.

A significant number of community contributors have been driving change, making commands more usable and useful, adding new features, and fixing bugs.

Several core commands have fallen by the wayside while these changes were being made. The reasons for this vary; in some cases, the commands are fundamentally incompatible with .NET Core. In a few cases, the commands were under restricted licensing agreements and could not be made open source. These differences are highlighted in this book and alternatives are demonstrated where possible.

Despite all this change, PowerShell still maintains a strong backward-compatibility. Lessons learned using Windows PowerShell can be applied to PowerShell Core and 7, and they will continue to be applicable to future versions of PowerShell.

This book is split into several sections. Much of this book is intended to act as a reference. The following topics will be covered:

- Exploring PowerShell fundamentals
- Working with data
- Automating with PowerShell
- Extending PowerShell

In the first section of this book, while exploring the PowerShell fundamentals, you will look at the use of language and cover as many building blocks as possible.

This chapter explores a diverse set of topics:

- What is PowerShell?
- PowerShell editors
- Getting help
- Command naming
- Command discovery
- Parameters, values, and parameter sets
- Introduction to providers
- Introduction to splatting
- Parser modes

- Experimental features

> **Technical requirements**
>
> This chapter makes use of the following on the Windows platform: PowerShell 7

Let's begin with what PowerShell is.

What is PowerShell?

PowerShell is a mixture of a command-line interface, a functional programming language, and an object-oriented programming language. PowerShell is based on Microsoft .NET, which gives it a level of open flexibility that was not available in Microsoft's scripting languages (such as VBScript or batch) before this.

PowerShell has been written to be highly discoverable. It has substantial built-in help that's accessible within the console via the `Get-Help` command. PowerShell has commands such as `Get-Member` to allow a user to discover the details of the values it returns.

PowerShell 7 can be installed alongside Windows PowerShell. Windows PowerShell is installed in `Windows\System32` by the Windows Management Framework packages, and it cannot be moved elsewhere. PowerShell Core and 7 are both installed in the `Program Files` folder and do not share any of the files used by Windows PowerShell. Preview versions of PowerShell can be installed alongside the full releases and have separate folder structures.

PowerShell is a complex language; a good editor can save time finding the right syntax to use in a script.

PowerShell editors

While you can develop PowerShell scripts using the Notepad application alone, it is rarely desirable. Using an editor that was designed to work with PowerShell can save a lot of time.

Specialized PowerShell editors such as Visual Studio Code (VS Code) and PowerShell Studio offer automatic completion (IntelliSense). IntelliSense reduces the amount of cross-referencing help content required while writing code. Finding a comfortable editor early on is a good way to ease into PowerShell; memorizing commands and parameters is not necessary.

In addition to the aforementioned editors, Windows PowerShell comes with PowerShell ISE. PowerShell ISE has not been updated for PowerShell 6 and higher and will only function correctly for Windows PowerShell.

PowerShell Studio is not free but includes graphical user interface development features.

VS Code is a highly recommended editor for PowerShell as it is free. VS Code is an open-source editor that was published by Microsoft and can be downloaded from `http://code.visualstudio.com`.

The functionality of VS Code can be enhanced by using extensions from the marketplace: `https://marketplace.visualstudio.com/VSCode`. The Extension installer is part of the VS Code user interface, and the types of available extensions are shown in *Figure 1.1*:

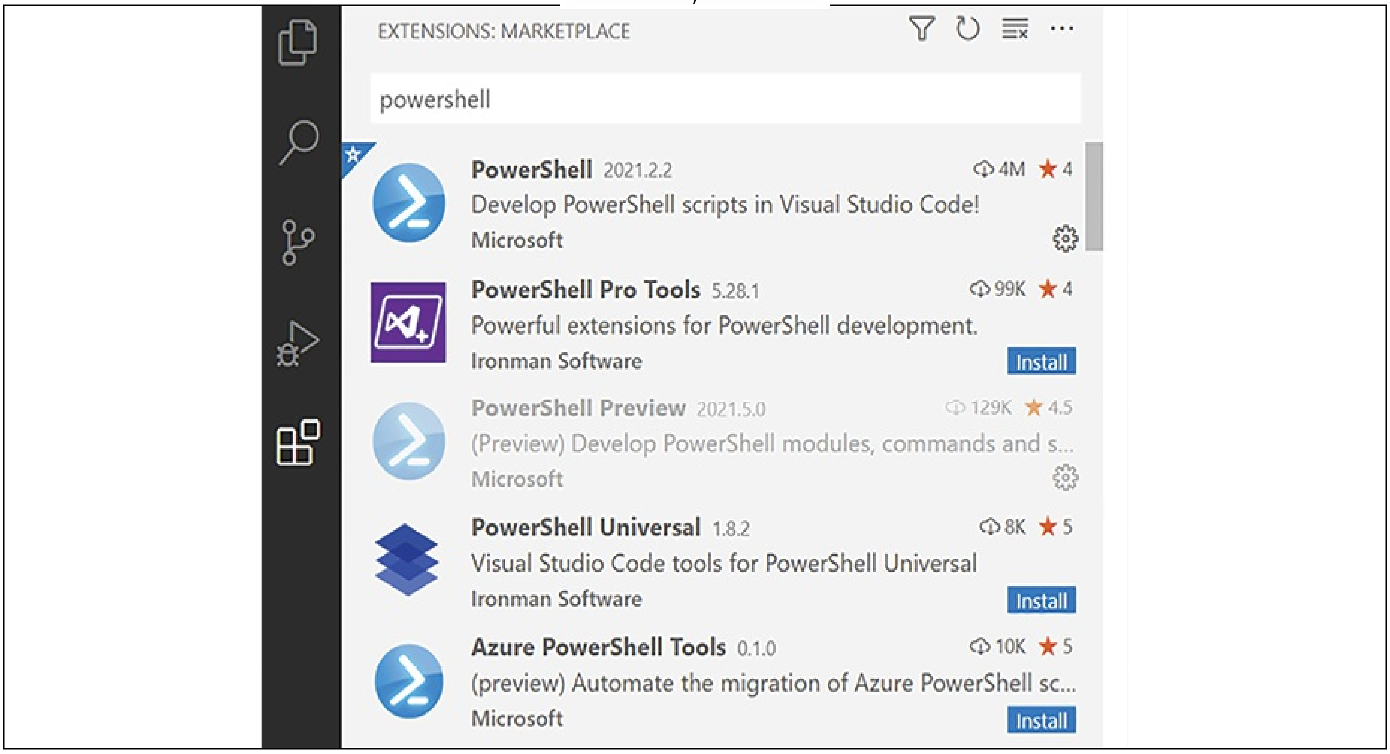


Figure 1.1: PowerShell extensions in VS Code

The icons available on the left-hand side change depending on the extensions installed. A new installation of VS Code will show fewer icons than *Figure 1.1*.

The PowerShell Extension should be installed. Other popular extensions include:

- Bracket Pair Colorizer 2
- Chocolatey
- Live Share
- Prettify JSON

Paid-for extensions, such as PowerShell Pro Tools, offer us the ability to design user interfaces in VS Code.

The integrated console in VS Code can be used with all installed versions of PowerShell. The following screenshot shows how to change the version of PowerShell used when editing a script. Note the clickable version in the bottom-right corner:

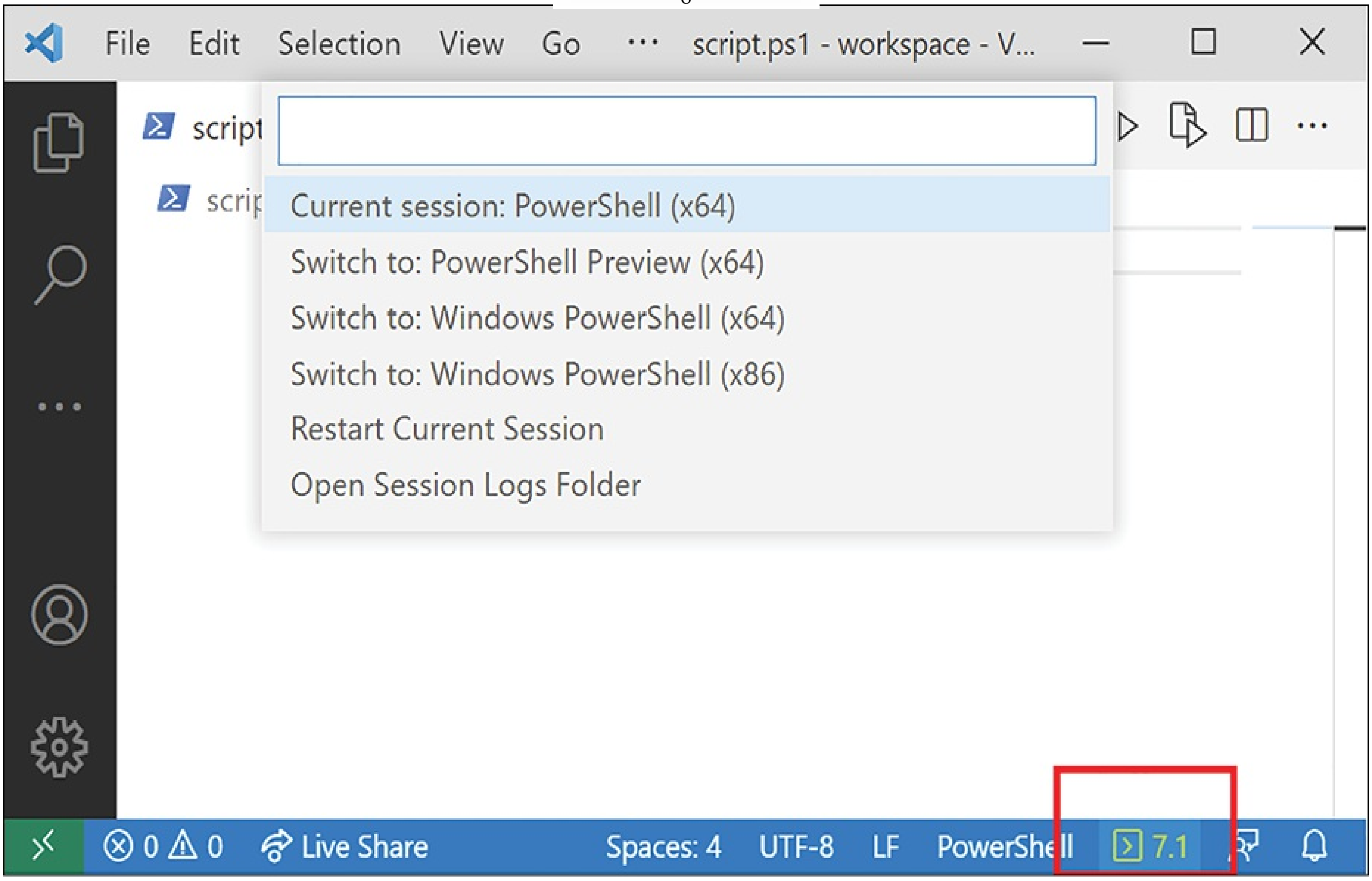


Figure 1.2: Choosing a PowerShell version

The IntelliSense version provided by the editor will list and hint at the possible commands and parameters available. Help content is available to fill in the details.

Getting help

PowerShell includes a built-in help system accessible using the `Get-Help` command. Gaining confidence using the built-in help system is an important part of working with PowerShell. Script authors and PowerShell developers can easily write their own help content when working with functions, scripts, and script modules.

Updatable help

The concept of updatable help was added in Windows PowerShell 3. It allows users to obtain the most recent versions of their help documentation outside of PowerShell on a web service. Help content can be downloaded and installed using the `Update-Help` command in PowerShell.

> **Which modules support updatable help?**
>
> You can view a list of modules that support updatable help by running the following command:
>
> ```
> Get-Module -ListAvailable | Where-Object HelpInfoURI
> ```

Help for the core components of PowerShell is not part of the PowerShell 7 installation package. Content must be downloaded before it can be viewed. The first time `Get-Help` is run, PowerShell prompts to update help. `Update-Help` requires write access to shared module directories such as `Program Files`. In Windows, this means using "Run as Administrator" to open a PowerShell console.

Computers with no internet access or computers behind a restrictive proxy server may not be able to download the help content directly. You need to use the `Save-Help` command, which is discussed later in this section, to work around this problem.

If PowerShell is unable to download help, it can only show a small amount of information about a command; for example, without downloading help, the content for the `Out-Null` command is minimal, as shown here:

```
PS> Get-Help Out-Null
NAME
    Out-Null

SYNTAX
    Out-Null [-InputObject <psobject>] [<CommonParameters>]

ALIASES
    None

REMARKS
    Get-Help cannot find the Help files for this cmdlet on this computer.
    It is displaying only partial help.
        -- To download and install Help files for the module that
           includes this cmdlet, use Update-Help.
        -- To view the Help topic for this cmdlet online, type:
           "Get-Help Out-Null -Online" or go to
           http://go.microsoft.com/fwlink/?LinkID=113366.
```

The help content in the preceding example is automatically generated by PowerShell. PowerShell inspects the command to determine which parameters are available.

Updatable help as a `help` file can be viewed using the following command:

```
Get-Help about_Updatable_Help
```

Updateable help is not entirely free from issues. Internet resources change as content moves around over time, which may invalidate `HelpInfoUri`, the URL stored within the module and used to retrieve help files. For example, the `ThreadJob` module briefly had help available via `Update-Help`; help for the `ThreadJob` module was not available when this chapter was written.

The Get-Help command

When `Get-Help` is used without parameters, it shows introductory help about the help system. This content is taken from the default `help` file (`Get-Help default`); a snippet of this is as follows:

```
PS> Get-Help
TOPIC
 Windows PowerShell Help System
SHORT DESCRIPTION
 Displays help about Windows PowerShell cmdlets and concepts.
LONG DESCRIPTION
 Windows PowerShell Help describes Windows PowerShell cmdlets,
```

> **Help content can be long**
>
> The help content, in most cases, does not fit on a single screen. The `help` command differs from `Get-Help` in that it pauses (waiting for a key to be pressed) after each page; for example:
>
> ```
> help default
> ```

The previous command is equivalent to running `Get-Help` and piping it into the `more` command:

```
Get-Help default | more
```

Alternatively, `Get-Help` can be asked to show a window:

```
Get-Help default -ShowWindow
```

The available help content may be listed using either of the following two commands:

```
Get-Help *
Get-Help -Category All
```

Help for a subject can be viewed as follows:

```
Get-Help -Name <Topic>
```

For example, help for the `Get-Variable` command may be shown:

```
Get-Help Get-Variable
```

If a help document includes an online version link, it may be opened in a browser with the `-Online` parameter:

```
Get-Help Get-Command -Online
```

The URL used for online help can be viewed using `Get-Command`:

```
Get-Command Get-Command | Select-Object HelpUri
```

The help content is broken down into several sections: name, synopsis, syntax, description, related links, and remarks.

Name simply includes the name of the command. Synopsis is a short description of the functionality provided by the command, often no more than one sentence. Description often provides much greater detail than synopsis. Related links and remarks are optional fields, which may include links to related content.

Syntax is covered in the following section in more detail as it is the most complex.

Syntax

The syntax section lists each of the possible combinations of parameters a command accepts; each of these is known as a **parameter set**.

A command that has more than one parameter set is shown by this example for the `Get-Process` command:

```
SYNTAX
    Get-Process [[-Name] <String[]>] [-ComputerName <String[]>]
    [-FileVersionInfo] [-Module] [<CommonParameters>]
    Get-Process [-ComputerName [<String[]>]] [-FileVersionInfo]
    [-Module] -InputObject <Process[]> [<CommonParameters>]
```

The syntax elements written in square brackets are optional; for example, syntax help for `Get-Process` shows that all its parameters are optional, as the following code shows:

```
SYNTAX
    Get-Process [[-Name] <String[]>] [-ComputerName <String[]>] [-FileVersionInfo] [-Module]
[<CommonParameters>]
```

As the `Name` parameter is optional, `Get-Process` may be run without any parameters. The command may also be run by specifying a value only and no parameter name. Alternatively, the parameter name can be included as well as the value.

Each of the following examples is a valid use of `Get-Process`:

```
Get-Process
Get-Process pwsh
Get-Process -Name pwsh
```

Get-Command can show syntax

`Get-Command` may be used to view the syntax for a command. For example, running the following command will show the same output as seen in the *Syntax* section of `Get-Help`:

```
Get-Command Get-Variable -Syntax
```

The different parameter types and how they are used are explored later in this chapter.

Examples

The Examples section of help provides working examples of how a command may be used. Help often includes more than one example.

In some cases, a command is sufficiently complex that it requires a detailed example to accompany parameter descriptions; in others, the command is simple, and a good example may serve in lieu of reading the help documentation.

PowerShell users can update help

Help documentation for built-in commands is open source. If a cmdlet is missing helpful examples, they can be added.

A link to the PowerShell-Docs repository is available at the bottom of the online help page. It should send a user to the en-US version of help:

https://github.com/MicrosoftDocs/PowerShell-Docs

Help for other languages is available under the same GitHub organization; for example:

https://github.com/MicrosoftDocs/powerShell-Docs.pl-pl

Some help for other languages may be generated by translation software.

Examples for a command can be requested by specifying the `Examples` parameter for `Get-Help`, as shown in the following example:

```
Get-Help Get-Process -Examples
```

The help information for most cmdlets usually includes several examples of its use, especially if the command has more than one parameter set.

Parameter

Parameters in PowerShell are used to supply named arguments to PowerShell commands.

Help for specific parameters can be requested as follows:

```
Get-Help Get-Command -Parameter <ParameterName>
```

The `Parameter` parameter allows for the quick retrieval of specific help for a single parameter; for example, help for the `Path` parameter of the `Import-Csv` command may be viewed:

```
PS> Get-Help Import-Csv -Parameter Path
-Path [<String[]>]
    Specifies the path to the CSV file to import. You can also pipe
    a path to `Import-Csv`.

    Required? false
    Position? 1
    Default value None
    Accept pipeline input? true (ByValue)
    Accept wildcard characters? false
```

The preceding content describes the parameter, whether the parameter is mandatory (`Required`), its position, default value, pipeline behavior, and support for wildcards.

Detailed and Full switches

The `Detailed` switch parameter (a parameter that does not require an argument) asks `Get-Help` to return the most help content.

The default sections returned by help are Name, Synopsis, Syntax, Description, Related Links, and Remarks.

When `Detailed` is requested, Parameters and Examples are added, and Related Links is excluded.

The `Detailed` parameter is used as follows:

```
Get-Help Get-Process -Detailed
```

The `Full` switch parameter includes all the default sections, as well as Parameters, Inputs, Outputs, Notes, and Examples.

The following code shows the sections detailing the input and output types for `Get-Process` from the full help document; content before those sections has been removed from this example:

```
PS> Get-Help Get-Process -Full
... <content removed> ...
INPUTS
    System.Diagnostics.Process

        You can pipe a process object to Get-Process.
OUTPUTS
    System.Diagnostics.Process, System.Diagnotics.FileVersionInfo, System.Diagnostics.ProcessModule

        By default, this cmdlet returns a System.Diagnostics.Process
        object. If you use the FileVersionInfo parameter, it returns
        a System.Diagnostics.FileVersionInfo object. If you use the
        Module parameter, without the FileVersionInfo parameter, it
        returns a System.Diagnostics.ProcessModule object.
```

`INPUTS` is typically used to describe the value types that can be piped to a command. Pipelines are introduced in *Chapter 3, Working with Objects in PowerShell*.

In addition to the extra sections, the `Full` switch parameter includes metadata in the parameter section, the same metadata seen when using `Get-Help Get-Process -Parameter Path`.

Help content in PowerShell is extensive and a valuable resource to have on any system running PowerShell.

Save-Help

The `Save-Help` command can be used with modules that support updatable help. It allows help content for installed modules to be saved to disk.

Help for a module can be downloaded using the following command. The destination folder must exist before running the

command:

```
Save-Help -Module Microsoft.PowerShell.Management -DestinationPath C:\PSHelp
```

By default, `Save-Help` attempts to download help content for the current UI culture; that is, the current user interface language. You can use the `Get-UICulture` command to view the values, as the following example shows:

```
PS> Get-UICulture
LCID             Name             DisplayName
----             ----             -----------
2057             en-GB            English (United Kingdom)
```

If help content is not available for that culture, `Save-Help` will not do anything and will not raise an error. For UI cultures other than `en-US`, the `C:\PSHelp` folder will likely be empty.

`Save-Help` can be instructed to download help in a specific language by using the `UICulture` parameter:

```
Save-Help -Module Microsoft.PowerShell.Management -DestinationPath C:\PSHelp -UICulture en-US
```

If help content is available, it is downloaded as shown in the `C:\PSHelp` folder here:

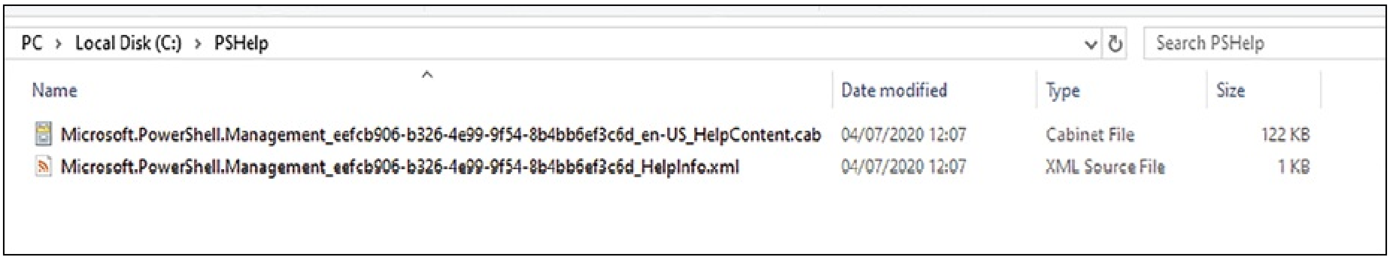


Figure 1.3: Downloaded help content for en-US

By default, `Save-Help` (and `Update-Help`) will not download help content more often than once every 24 hours. This can be seen using the `Verbose` switch parameter:

```
PS> Save-Help -Module Microsoft.PowerShell.Management -DestinationPath C:\PSHelp -UICulture en-US -
Verbose
VERBOSE: Help was not saved for the module Microsoft.PowerShell.Management, because the Save-Help
command was run on this computer within the last 24 hours.
```

The `Verbose` switch parameter is used to make any verbose messages the command author has included visible in the console.

If help content is available for other cultures, and that content is downloaded immediately after `en-US`, then you must add the `Force` parameter:

```
Save-Help -Module Microsoft.PowerShell.Management -DestinationPath C:\PSHelp -UICulture pl-PL -Force
```

However, as help content for the `Microsoft.PowerShell.Management` module is only available in `en-US`, the preceding command displays an error message describing which cultures help is available for.

Help content for all modules supporting updateable help can be saved as follows:

```
Save-Help -DestinationPath C:\PSHelp -UICulture en-US
```

Saved help content can be copied to another computer and imported using `Update-Help`. This technique is useful for computers that do not have internet access as it means help content can be made available.

Update-Help

The `Update-Help` command performs two tasks:

- Update help files on the local computer from the internet
- Updates help files on the local computer from previously saved `help` files

To update help from the internet for all modules that support updateable help, run the `Update-Help` cmdlet with no parameters:

```
Update-Help
```

Administrative rights are required for Update-Help

Updating help for some modules requires administrative rights (run as administrator). This applies to modules that are stored in protected areas of the filesystem, such as those in `Program Files`.

For UI cultures other than `en-US`, the `UICulture` parameter may be required:

```
Update-Help -UICulture en-US
```

Like `Save-Help`, `Update-Help` will not download help for a module more than once every 24 hours by default. This can be overridden by using the `Force` parameter:

```
Update-Help -Name Microsoft.PowerShell.Management -Force -UICulture en-US
```

Help content that was saved using `Save-Help` can be imported from a folder using the `SourcePath` parameter:

```
Update-Help -SourcePath C:\PSHelp
```

If the folder does not contain content for the current UI culture (shown with `Get-UICulture`), an error message will be displayed:

```
PS> Update-Help -Module Microsoft.PowerShell.Management -SourcePath C:\PSHelp
Update-Help: Failed to update Help for the module(s) 'Microsoft.PowerShell.Management' with UI
culture(s) {en-GB} : Unable to retrieve the HelpInfo XML file for UI culture en-GB. Make sure the
HelpInfoUri property in the module manifest is valid or check your network connection and then try the
command again..
```

You can use the `UICulture` parameter again to update help content from the folder:

```
Update-Help -Module Microsoft.PowerShell.Management -SourcePath C:\PSHelp -UICulture en-US
```

Help content is not limited to help for specific commands. PowerShell includes a large number of topical help documents.

About_* help files

`About_*` documents describe features of a language or concepts that apply to more than one command. These items do not fit into help for individual commands.

The list of help files can be viewed by running `Get-Help` with the category set to `HelpFile`, as demonstrated in the following code:

```
Get-Help -Category HelpFile
```

Alternatively, wildcards can be used with the `Name` parameter of `Get-Help`:

```
Get-Help -Name About_*
```

These help files cover a huge variety of topics from aliases, to modules, to WMI. A number of these are shown here. The list will vary, depending on the modules installed on the computer running the command:

```
Name                          Category  Module     Synopsis
----                          --------  ------     --------
default                       HelpFile             SHORT DESCRIPTION
about_PSReadLine              HelpFile
about_Configuration           HelpFile             The Configuration...
about_Aliases                 HelpFile
about_Alias_Provider          HelpFile
about_Arithmetic_Operators    HelpFile
about_Arrays                  HelpFile
about_Assignment_Operators    HelpFile
about_Automatic_Variables     HelpFile
about_Break                   HelpFile
```

Using help content is an important part of working with PowerShell. Memorizing content is not necessary where instructions and reference material are easily accessible.

`Get-Help` may lead to finding a command to help achieve a task; however, it is often quicker to search using `Get-Command`.

Command naming and discovery

Commands in PowerShell are formed around verb and noun pairs in the form verb-noun.

This feature is useful when finding commands; it allows you to make educated guesses so that there is little need to memorize long lists of commands.

Verbs

The list of verbs is maintained by Microsoft. Verbs are words such as `Add`, `Get`, `Set`, and `New`. This formal approach to naming commands greatly assists in discovery.

You can view the verbs available in PowerShell using the following command:

```
Get-Verb
```

Verbs are grouped around different areas, such as data, life cycle, and security. Complementary actions such as encryption and decryption tend to use verbs in the same group; for example, the verb `Protect` may be used to encrypt something and the verb `Unprotect` may be used to decrypt something.

> **Verb descriptions**
>
> A detailed list of verbs, along with their use cases, is available on MSDN:
>
> https://docs.microsoft.com/en-us/powershell/scripting/developer/cmdlet/approved-verbs-for-windows-powershell-commands?view=powershell-7

It is possible, although not recommend, to use verbs other than those in the approved list. If a command with an unapproved verb is written and included in a module, a warning message will be displayed every time the module is imported.

Nouns

A noun provides a very short description of the object the command is expecting to act on. The noun part may be a single word,

as is the case with `Get-Process`, `New-Item`, or `Get-Help`, or more than one word, as seen with `Get-ChildItem`, `Invoke-WebRequest`, or `Send-MailMessage`.

Command names often include a prefix on the noun. For example, the `Get-ADUser` command is part of the `ActiveDirectory` module. Modules are explored in *Chapter 2*, *Modules and Snap-Ins*. All commands in the `ActiveDirectory` module use `AD` to prefix the noun.

Finding commands

The verb-noun pairing strives to make it easier to find commands without resorting to search engines.

For example, if the goal was to list firewall rules, the following command may be used to show the `Get` commands that might affect the firewall:

```
PS> Get-Command Get-*Firewall*
CommandType Name                                Version Source
----------- ----                                ------- ------
Function    Get-NetFirewallAddressFilter        2.0.0.0 NetSecurity
Function    Get-NetFirewallApplicationFilter    2.0.0.0 NetSecurity
Function    Get-NetFirewallInterfaceFilter      2.0.0.0 NetSecurity
Function    Get-NetFirewallInterfaceTypeFilter 2.0.0.0 NetSecurity
Function    Get-NetFirewallPortFilter           2.0.0.0 NetSecurity
Function    Get-NetFirewallProfile              2.0.0.0 NetSecurity
Function    Get-NetFirewallRule                 2.0.0.0 NetSecurity
Function    Get-NetFirewallSecurityFilter       2.0.0.0 NetSecurity
Function    Get-NetFirewallServiceFilter        2.0.0.0 NetSecurity
Function    Get-NetFirewallSetting              2.0.0.0 NetSecurity
```

The list of commands returned may vary, depending on the modules installed on the computer.

From the preceding list, `Get-NetFirewallRule` closely matches the requirement (to see a list of firewall rules) and should be explored.

Once a potential command has been found, `Get-Help` can be used to assess whether the command is suitable.

Aliases

An alias in PowerShell is an alternate name for a command. A command may have more than one alias. Unlike languages like Bash, an alias cannot include parameters. The use of aliases in production code is generally discouraged.

The list of aliases can be viewed by using `Get-Alias`. The first few aliases are shown in the following example:

```
PS> Get-Alias
CommandType Name
----------- ----
Alias       % -> ForEach-Object
Alias       ? -> Where-Object
Alias       ac -> Add-Content
Alias       cat -> Get-Content
Alias       cd -> Set-Location
```

`Get-Alias` may be used to find the command behind an alias:

```
Get-Alias dir
```

`Get-Alias` may also be used to find the aliases for any command:

```
PS> Get-Alias -Definition Get-ChildItem
CommandType     Name                      Version    Source
-----------     ----                      -------    ------
Alias           dir -> Get-ChildItem
Alias           gci -> Get-ChildItem
Alias           ls -> Get-ChildItem
```

Examples of aliases that are frequently used in examples on the internet include the following:

- `%` for `ForEach-Object`
- `?` for `Where-Object`
- `cd` for `Set-Location`
- `gc` or `cat` for `Get-Content`
- `ls` or `dir` for `Get-ChildItem`
- `man` for `help` (and then `Get-Help`)

An alias does not change how a command is used. There is no difference in the result of the following two commands:

```
cd $env:TEMP
Set-Location $env:TEMP
```

New aliases are created with the `New-Alias` command. For example, an alias named `grep` for the `Select-String` command can be created as follows:

```
New-Alias grep -Value Select-String
```

Aliases can be removed using the `Remove-Alias` command, including default aliases such as `ls`:

```
Remove-Alias grep
```

Aliases may also be removed using `Remove-Item` as an alternative to `Remove-Alias`:

```
Remove-Item alias:\grep
```

Aliases created in one session are not remembered when a new PowerShell session is started.

> More information is available about aliases in the help file `about_Aliases`. You can view this help file using the following command:
>
> ```
> Get-Help about_Aliases
> ```

Commands (and aliases) use parameters to pass arguments into a command.

Parameters, values, and parameter sets

As seen while looking at syntax in `Get-Help`, commands accept a mixture of parameters. The following sections show how these parameters are described in help and how to use them.

Parameters

When viewing help for a command, several different conventions are used to describe when a parameter is required and how it should be used. These conventions include:

- Optional parameters, where parameter names and arguments are enclosed in a single pair of square brackets.
- Optional positional parameters, the same as an optional parameter but with the parameter name also enclosed in square brackets.
- Mandatory parameters, where the parameter name and argument are not bracketed.
- Mandatory positional parameters, where the parameter name is in square brackets, but the argument is not.

The following sections show each of these conventions in greater detail.

Optional parameters

Optional parameters are surrounded by square brackets. If a parameter is used, a value (or argument) must be supplied:

```
SYNTAX
    Get-Process [-ComputerName <String[]>] ...
```

If a value for the `ComputerName` parameter is to be used, the name of the parameter must also be specified. This is shown in the following example:

```
Get-Process -ComputerName somecomputer
```

The preceding command requests the list of processes on the computer, `somecomputer`.

Optional positional parameters

An optional positional parameter is surrounded by square brackets, like an optional parameter. In addition, the parameter name itself is enclosed in square brackets. This indicates that the parameter it optional and that if it is used, the parameter and value can be supplied, or just the value without the parameter name.

It is not uncommon to see an optional positional parameter as the first parameter:

```
SYNTAX
    Get-Process [[-Name] <String[]>] ...
```

In this example, either of the following may be used:

```
Get-Process -Name powershell
Get-Process powershell
```

The output from the two commands is identical. This includes the parameter name, which, even when it is optional, is less ambiguous and therefore a recommended practice.

Mandatory parameters

A mandatory parameter must always be supplied and is written as follows:

```
SYNTAX
    Get-ADUser -Filter <string> ...
```

In this case, the `Filter` parameter name must be used, and it must be given a value. For example, to supply a `Filter` for the command, the `Filter` parameter must be explicitly written:

```
Get-ADUser -Filter 'sAMAccountName -eq "SomeName"'
```

The `Get-ADUser` command has a second parameter set that uses a different parameter name with a positional value.

Mandatory positional parameters

Mandatory parameters must always be supplied, but in some cases, it is possible to supply the value without using the parameter name. The parameter the value applies to is based on position.

Parameters that are mandatory and accept values based on position are written with the parameter name only in square brackets, as shown here:

```
SYNTAX
    Get-ADUser [-Identity] <ADUser> ...
```

In this case, the `Identity` parameter name is optional, but the value is not. This command may be used as described by either of the following examples:

```
Get-ADUser -Identity useridentity
Get-ADUser useridentity
```

In both cases, the supplied value fills the `Identity` parameter.

The `Add-Content` command has a parameter set that uses more than one mandatory positional parameter. The first part of the syntax for the parameter set is shown here:

```
Add-Content [-Path] <String[]> [-Value] <Object[]>
```

In this case, the command may be called using any of the following:

```
Add-Content -Path c:\temp\file.txt -Value 'Hello world'
Add-Content -Value 'Hello world' -Path c:\temp\file.txt
Add-Content 'Hello world' -Path c:\temp\file.txt
Add-Content c:\temp\file.txt -Value 'Hello world'
Add-Content c:\temp\file.txt 'Hello world'
```

The first of these is easiest to read as both parameters are explicitly named and tends to be the better style to use.

Each of the parameters so far have required an argument, a value. PowerShell also allows parameters that do not require arguments.

Switch parameters

A switch parameter does not require an argument. If the switch is present, the value is equivalent to true, while if the switch parameter is absent, it is equivalent to false.

As with the other types of parameters, optional use is denoted by using square brackets around the parameter.

Switch parameters are typically used to toggle a behavior on. For example, `Recurse` is a switch parameter for `Get-ChildItem`:

```
SYNTAX
    Get-ChildItem ... [-Recurse] ...
```

Using the switch instructs `Get-ChildItem` to recurse when listing the content of a directory, as shown here:

```
Get-ChildItem c:\windows -Recurse
```

It is possible to supply a value for a switch parameter from a variable. This might be desirable if writing a script where the presence of a switch parameter is based on another variable. As switch parameters do not normally expect a value, a syntax change is required:

```
# Code which determines if Recurse is required
$recurse = $false
Get-ChildItem c:\windows -Recurse:$recurse
```

In some cases, a switch parameter will default to present, and it may be desirable to stop the parameter applying. The most common example is the `Confirm` parameter, which will be explored later in this chapter.

Parameter values

The syntax blocks explored in the preceding sections show the type that is expected when providing a value for a parameter. A type is a .NET concept; it describes what a value is, how it behaves, and what it can do. Types will be covered in greater detail in *Chapter 7, Working with .NET*.

The `Get-CimInstance` command expects a string as the argument for the `ClassName` parameter. This is shown in the snippet taken from the syntax block:

```
Get-CimInstance [-ClassName] <String>
```

A string is a sequence of characters. For example, the string `Win32_Service` can be used as follows:

```
Get-CimInstance -ClassName Win32_Service
```

`ClassName` must always be a single value. If more than one value is supplied, an error will be displayed:

```
PS> Get-CimInstance -ClassName Win32_Service, Win32_Process
Get-CimInstance: Cannot convert 'System.Object[]' to the type 'System.String' required by parameter
'ClassName'. Specified method is not supported.
```

Parameters that accept more than one value use `[]` after the type name. This indicates that the type is an array. The `Name` parameter for the `Get-Service` command is shown here:

```
Get-Service [[-Name] <String[]>]
```

In this case, the parameter type is an array of strings. An array may consist of one or more strings separated by a comma:

```
PS> Get-Service -Name WinDefend, WlanSvc
Status   Name               DisplayName
------   ----               -----------
Running  WinDefend          Windows Defender Antivirus Service
Running  WlanSvc            WLAN AutoConfig
```

PowerShell will attempt to coerce any value supplied into the required type. A single string can be used as an argument for the parameter. PowerShell will convert the single value into an array of strings with one element; for example:

```
Get-Service -Name WinDefend
```

Each of the commands used in this section will allow the value to be entered without the parameter name. For example, for `Get-Service`, the `Name` parameter can be omitted:

```
Get-Service WinDefend
Get-Service WinDefend, WlanSvc
```

When using positional parameters, PowerShell can use the type to determine which parameter (and which parameter set) should be used.

Parameter sets

In PowerShell, a parameter set is a set of parameters that may be used together when running a command.

Many of the commands in PowerShell have more than one parameter set. This was seen when looking at the `Syntax` section when using `Get-Help`.

For example, the `Stop-Process` command has three parameter sets:

```
SYNTAX
    Stop-Process [-Id] <Int32[]> [-Confirm] [-Force] [-PassThru] [-WhatIf] [<CommonParameters>]

    Stop-Process [-InputObject] <Process[]> [-Confirm] [-Force] [-PassThru] [-WhatIf]
[<CommonParameters>]

    Stop-Process [-Confirm] [-Force] -Name <String[]> [-PassThru] [-WhatIf] [<CommonParameters>]
```

PowerShell will attempt to find a matching parameter set based on the parameters and values it is given.

The parameter sets for `Stop-Process` have two different sets that will accept a value by position:

```
Stop-Process [-Id] <Int32[]>
Stop-Process [-InputObject] <Process[]>
```

The first expects an ID as an integer. The second expects a `Process` object, an object returned by the `Get-Process` command.

The variable `$PID` is an automatic variable that holds the process ID (an integer) of the current PowerShell console. Running the following command will stop the PowerShell process. The first parameter set for `Stop-Process` is chosen because an integer value is used:

```
Stop-Process $PID
```

The second parameter set expects a value for `InputObject`. Again, this may be supplied as a positional parameter (or via the pipeline). In this case, PowerShell distinguishes based on its type. The following snippet contains the three possible approaches available when using the `InputObject` parameter:

```
$process = Start-Process notepad -PassThru
Stop-Process -InputObject $process
Stop-Process $process
$process | Stop-Process
```

Pipeline input

`Get-Help` shows which parameters accept pipeline input in the help for each parameter. This may be viewed using either of the following commands:

```
Get-Help Stop-Process -Parameter *
Get-Help Stop-Process -Full
```

Examples are likely to show how to use the parameters with a pipeline.

If `Get-Help` is incomplete, `Get-Command` can be used to explore parameters:

```
(Get-Command Stop-Process).Parameters.InputObject.Attributes
```

Each of the parameter sets here also shows that the command supports common parameters.

Common parameters

Common parameters are used to control some of the standardized functionality PowerShell provides, such as verbose output and actions to take when errors occur.

When looking at the syntax, most commands will end with a `CommonParameters` item:

```
SYNTAX
    Get-Process ... [<CommonParameters>]
```

The following is a list of common parameters:

- `Debug`
- `ErrorAction`
- `ErrorVariable`
- `InformationAction`
- `InformationVariable`
- `OutBuffer`
- `OutVariable`
- `PipelineVariable`
- `Verbose`
- `WarningAction`
- `WarningVariable`

Each is described in the `about_CommonParameters` document:

```
Get-Help about_CommonParameters
```

The help document is also available online: https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_commonparameters?view=powershell-7.

For example, `Stop-Process` does not explicitly state that it has a `Verbose` switch parameter, but since `Verbose` is a common parameter, it may be used. This can be seen if `notepad` is started and immediately stopped:

```
PS> Start-Process notepad -Verbose -PassThru | Stop-Process -Verbose
VERBOSE: Performing the operation "Stop-Process" on target "notepad (5592)".
```

> **Not so verbose**
>
> Just because a command supports common parameters does not mean it uses them. For example, `Get-Process` supports the `Verbose` parameter, yet it does not write any verbose output when `Verbose` is specified.

In addition to common parameters, PowerShell also offers specialized parameters for commands that make changes.

Confirm and WhatIf

`Confirm` and `WhatIf` can be used with commands that make changes to files, variables, data, and so on. These parameters are often used with commands that use the verbs `Set` or `Remove`, but the parameters are not limited to specific verbs.

`Confirm` and `WhatIf` have associated preference variables that are used to customize default behavior in PowerShell. Preference variables have an `about` file, which may be viewed using the following command:

```
Get-Help about_Preference_Variables
```

Confirm and ConfirmPreference

You can use the `Confirm` switch parameter and the `ConfirmPreference` variable to decide if a command should prompt. The decision to prompt is based on a comparison of `ConfirmPreference` with `ConfirmImpact` when set by a command author.

`ConfirmPreference` has four possible values:

- `High` : Prompts when command impact is `High` (default)
- `Medium` : Prompts when command impact is `Medium` or `High`
- `Low` : Prompts when command impact is `Low` , `Medium` , or `High`
- `None` : Never prompts

`ConfirmImpact` uses the same four values.

In Windows PowerShell, the default value for `ConfirmImpact` is `Medium` .

In PowerShell 7, the default value for `ConfirmImpact` is `None` . If the command uses `SupportsShouldProcess` , then the default is `Medium` . `SupportsShouldProcess` is explored in greater detail in *Chapter 17, Scripts, Functions, and Script Blocks*.

Finding commands with a specific impact

The following snippet returns a list of all commands that state they have a high impact:

```
Get-Command -CommandType Cmdlet, Function | Where-Object {
    $metadata = [System.Management.Automation.CommandMetadata]$_
    $metadata.ConfirmImpact -eq 'High'
}
```

If the `Confirm` parameter is explicitly provided, the value of `ConfirmPreference` within the scope of the command is set to `Low` , which will trigger any confirmation prompts. Scoping of preference variables is explored in greater detail in *Chapter 17, Scripts, Functions, and Script Blocks.*

The `Confirm` switch parameter therefore causes a command to prompt before an action is taken; for example, the `Confirm` switch parameter forces `Remove-Item` to prompt when a file is to be removed:

```
PS> Set-Location $env:TEMP
PS> New-Item FileName.txt -Force
PS> Remove-Item FileName.txt -Confirm

Confirm
Are you sure you want to perform this action?
Performing the operation "Remove File" on target "C:\Users\whoami\AppData\Local\Temp\FileName.txt".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"):
```

In the previous example, a confirmation prompt was explicitly requested. In a similar manner, confirmation prompts may be suppressed. For example, the value of the `Confirm` parameter may be explicitly set to `false` :

```
Set-Location $env:TEMP
New-Item FileName.txt -Force
Remove-Item FileName.txt -Confirm:$false
```

The ability to provide a value for the `Confirm` parameter is useful for commands that prompt by default; for example, `Clear-RecycleBin` prompts by default:

```
PS> Clear-RecycleBin

Confirm
Are you sure you want to perform this action?
Performing the operation "Clear-RecycleBin" on target " All of the contents of the Recycle Bin".
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"):
```

Setting the `Confirm` parameter to `false` for `Clear-RecycleBin` bypasses the prompt and immediately empties the recycle bin:

```
Clear-RecycleBin -Confirm:$false
```

If the `Confirm` parameter is not set, whether a prompt is shown is determined by PowerShell. The value of the `ConfirmPreference` variable is compared with `ConfirmImpact` on a command.

> **There is more than one way to prompt**
>
> There are two ways of requesting confirmation in PowerShell. These are `ShouldProcess` and `ShouldContinue`. These are explored in *Chapter 17, Scripts, Functions, and Script Blocks*.
>
> `ShouldProcess` is affected by the `Confirm` parameter and `ConfirmPreference` variable.
>
> `ShouldContinue` is a forced prompt and is unaffected by the `Confirm` parameter and `ConfirmPreference` variable.
>
> For example, `Remove-Item` will always prompt when attempting to delete a directory that is not empty without supplying the `Recurse` parameter.
>
> It is not possible to easily discover commands using forced prompts. Reviewing documentation and testing is vital.

By default, the value of `ConfirmPreference` is `High`. This means that prompts will be raised when `ConfirmImpact` for a command is `High`. The default value for `ConfirmPreference` may be viewed as follows:

```
PS> $ConfirmPreference
High
```

> **Finding ConfirmImpact**
>
> In scripts and functions, the `ConfirmImpact` setting is part of the `CmdletBinding` attribute:
>
> ```
> [CmdletBinding(ConfirmImpact = 'High')]
> ```
>
> If `CmdletBinding` or `ConfirmImpact` are not present, the impact is `Medium` in Windows PowerShell and `None` in PowerShell 7.
>
> The impact of a function or cmdlet may be viewed using the `ConfirmImpact` property of a command's metadata:
>
> ```
> [System.Management.Automation.CommandMetadata](Get-Command Remove-Item)
> ```
>
> The use of `CmdletBinding` is explored in detail in *Chapter 17, Scripts, Functions, and Script Blocks*.

A new value for `ConfirmPreference` may be set by assigning it in the console; for example, it can be set to `Low`. When the preference variable is set to `Low`, prompts may be raised by all commands where `ConfirmImpact` is `Low`, `Medium`, or `High`.

```
$ConfirmPreference = 'Low'
```

> **ConfirmPreference and the Confirm parameter**
>
> When `ConfirmPreference` is set to `None` to suppress confirmation prompts, confirmation may still be explicitly requested using the `Confirm` parameter; for example:
>
> ```
> $ConfirmPreference = 'None'
> New-Item NewFile.txt -Confirm
> ```

Support for confirmation also provides support for `WhatIf`.

WhatIf and WhatIfPreference

`WhatIf` is typically used when testing a command. If implemented correctly by a command author, `WhatIf` should allow a state changing command to be run without it making the change.

The `WhatIf` parameter has an associated preference variable, `WhatIfPreference`, which may be set to either `true` or `false`. The default value is `false`.

The `WhatIf` parameter replaces the confirmation prompt with a simple statement describing the action the command would have taken. Using `Remove-Item` as an example again, a message will be displayed, and the file will not be deleted:

```
PS> Set-Location $env:TEMP
PS> New-Item FileName.txt -Force
PS> Remove-Item FileName.txt -WhatIf

What if: Performing the operation "Remove File" on target
"C:\Users\whoami\AppData\Local\Temp\FileName.txt".
```

If both `Confirm` and `WhatIf` are used with a command, `WhatIf` takes precedence. `WhatIf` messages is shown, while confirmation is not requested.

`WhatIf` can be explicitly set on a per-command basis by supplying a value in the same manner as the `Confirm` parameter. For example, `WhatIf` might be explicitly set to `false`:

```
'Some message' | Out-File $env:TEMP\test.txt -WhatIf:$false
```

Setting `WhatIf` in the manner used here might, for instance, be useful if a log file should be written even if other state-changing commands are ignored.

If the preference variable is set to `true`, all commands that support `WhatIf` act as if the parameter is set explicitly. A new value may be set for the variable, as shown in the following code:

```
$WhatIfPreference = $true
```

The `WhatIf` preference variable takes precedence over the `Confirm` parameter. For example, the `WhatIf` dialog is shown when running the following `New-Item`, but the `Confirm` prompt is not:

```
PS> $WhatIfPreference = $true
PS> New-Item NewFile.txt -Confirm
What if: Performing the operation "Create File" on target "Destination:
C:\Users\whoami\AppData\Local\Temp\NewFile.txt".
```

Restarting the console will restore preference variables to their default values.

The behavior of `Confirm` and `WhatIf` is prescribed by PowerShell. Parameters such as `Force` and `PassThru` are commonly used in PowerShell but have less well-defined behavior.

Force parameter

The `Force` parameter is not one of the common parameters with behavior defined by PowerShell itself, but the parameter is frequently used.

`Force` has no fixed usage; the effect of using `Force` is a choice a command author must make. Help documentation should state the effect of using `Force` with a command. For example, the use of `Force` with `Remove-Item` is available:

```
Get-Help Remove-Item -Parameter Force
```

With the `Force` parameter, `New-Item` overwrites any existing file with the same path. When used with `Remove-Item`, the `Force` parameter allows the removal of files with `Hidden` or `System` attributes.

The error that is generated when attempting to delete a `Hidden` file is shown in the following code:

```
PS> Set-Location $env:TEMP
PS> New-Item FileName.txt -Force
PS> Set-ItemProperty FileName.txt -Name Attributes -Value Hidden
PS> Remove-Item FileName.txt
Remove-Item: You do not have sufficient access rights to perform this operation or the item is hidden,
system, or read
only.RemoveFileSystemItemUnAuthorizedAccess,Microsoft.PowerShell.Commands.RemoveItemCommand
```

Adding the `Force` parameter allows the operation to continue:

```
Remove-Item FileName.txt -Force
```

The `Force` parameter may be worth exploring if a command is prompting, and the prompts cannot be suppressed using the `Confirm` parameter or the `ConfirmPreference` variable.

PassThru parameter

The `PassThru` parameter, like `Force`, is frequently used, but the behavior of the parameter is not defined by PowerShell. However, `PassThru` tends to have predictable behavior.

The `PasThru` parameter is typically used with commands that do not normally generate output and is used to force the command to return the object it was working with.

For example, the `Start-Process` command does not normally return any output. If `PassThru` is used, it will return the process it created:

```
PS> Start-Process notepad -PassThru
 NPM(K)    PM(M)      WS(M)     CPU(s)      Id  SI ProcessName
 ------    -----      -----     ------      --  -- -----------
      9     1.98       6.70       0.05   22636   1 notepad
```

The `PassThru` parameter is therefore useful if more work is to be done with the object after the first command has finished.

For example, `PassThru` might be used with `Set-Service`, which ordinarily does not return output, allowing a service to be started immediately after another change:

```
Get-Service Audiosrv |
    Set-Service -StartupType Automatic -PassThru |
    Start-Service
```

Parameters in PowerShell is a complex topic, but are a vital part of working with the language.

Introduction to providers

A provider in PowerShell is a specialized interface to a service or dataset that presents items to the end user in the same style as a filesystem.

All operating systems include the following providers:

- **Alias**: PowerShell aliases
- **Environment**: Environment variables (for the process)
- **FileSystem**: Files and folders

- **Function**: Any functions in the session
- **Variable**: Any variables in the session

Windows operating systems also include Windows-specific providers:

- **Registry**: All loaded registry hives
- **Certificate**: The `LocalMachine` and `CurrentUser` certificate stores
- **WSMan**: Windows remoting configuration

A number of modules, such as the `ActiveDirectory` and `WebAdministration` modules, add service-specific providers when imported.

A longer description of `Providers` can be seen by viewing the `about` file:

```
Get-Help about_Providers
```

You can view the providers available in the current PowerShell session by running `Get-PSProvider`, as shown in the following example:

```
PS> Get-PSProvider

Name          Capabilities                       Drives
----          ------------                       ------
Registry      ShouldProcess, Transactions        {HKLM, HKCU}
Alias         ShouldProcess                      {Alias}
Environment   ShouldProcess                      {Env}
FileSystem    Filter, ShouldProcess, Credentials {C, D}
Function      ShouldProcess                      {Function}
Variable      ShouldProcess                      {Variable}
Certificate   ShouldProcess                      {Cert}
WSMan         Credentials                        {WSMan}
```

Each of the previous providers has a help file associated with it. In PowerShell, the help files are named `about_<ProviderName>_Provider`; for example:

```
Get-Help -Name about_Certificate_Provider
```

A list of all help files for providers in PowerShell 7 can be seen by running the following command:

```
Get-Help -Name About_*_Provider
```

In Windows PowerShell, the help files have a special category and are accessed using by name; for example:

```
Get-Help -Name Certificate -Category Provider
```

Or, the provider help files can be listed by category:

```
Get-Help -Category Provider
```

The provider-specific help documents describe the additional parameters added to `*-Item` and `*-ChildItem`, as well as `Test-Path`, `Get-Content`, `Set-Content`, `Add-Content`, `Get-Acl`, `Set-Acl`, and so on.

Provider-specific parameters, when added to the preceding commands, allow provider-specific values for filtering, making changes to existing items, and creating new items.

PowerShell offers tab completion for parameters when the `Path` parameter has been defined. For example, entering the following partial command, then pressing tab, will cycle through the parameters available to the Certificate provider:

```
Get-ChildItem -Path cert:\LocalMachine\Root -
```

For example, pressing Tab several times after the hyphen is entered offers up the `CodeSigningCert` parameter.

Missing parameters for the Certificate provider

Several parameters documented in the provider help file were temporarily removed from the certificate provider in PowerShell 6:

- ExpiringInDays
- DNSName
- DocumentEncryptionCert
- EKU
- SSLServerAuthentication

These are present in Windows PowerShell, but absent from PowerShell 6 to 7.0. The preceding parameters are present again in PowerShell 7.1.

The items within a provider can be accessed by following the provider name with two colons. For example, the content of the variable provider can be shown as follows:

```
Get-ChildItem variable::
```

The same approach can be used to view the top-level items available in the Registry provider on Windows:

```
Get-ChildItem registry::
```

Child items can be accessed by adding a name; for example, a variable:

```
Get-Item variable::true
```

The preceding command is equivalent to running `Get-Variable true`.

The `FileSystem` provider returns an error if you attempt to access `FileSystem::` without specifying a path. A child item must be specified; for example:

```
Get-ChildItem FileSystem::C:\Windows
```

While it is possible to access providers directly using the preceding notation, several of the providers are given names and are presented in the same manner as a Windows disk drive.

Drives and providers

Drives are labels used to access data from providers by name. Drives are automatically made available for `FileSystem` based on the drive letters used for mounted partition in Windows.

The output from `Get-PSProvider` in the previous section shows that each provider has one or more drives associated with it.

Alternatively, the list of drives can be seen using `Get-PSDrive`, as shown in the following example:

```
PS> Get-PSDrive
Name     Used (GB) Free (GB) Provider      Root
----     --------- --------- --------      ----
Alias                        Alias
C            89.13    111.64 FileSystem    C:\
Cert                         Certificate   \
D             0.45     21.86 FileSystem    D:\
Env                          Environment
Function                     Function
HKCU                         Registry      HKEY_CURRENT_USER
HKLM                         Registry      HKEY_LOCAL_MACHINE
Variable                     Variable
WSMan                        WSMan
```

As providers present data in a similar manner to a filesystem, accessing a provider is like working with a disk drive. This example shows how `Get-ChildItem` changes when exploring the `Cert` drive. The first few certificates are shown:

```
PS C:\> Set-Location Cert:\LocalMachine\Root
PS Cert:\LocalMachine\Root> Get-ChildItem
 Directory: Microsoft.PowerShell.Security\Certificate::LocalMachine\Root
Thumbprint                                Subject
----------                                -------
CDD4EEAE6000AC7F40C3802C171E30148030C072  CN=Microsoft Root Certif...
BE36A4562FB2EE05DBB3D32323ADF445084ED656  CN=Thawte Timestamping C...
A43489159A520F0D93D032CCAF37E7FE20A8B419  CN=Microsoft Root Author...
```

By default, drives are available for the current user, `HKEY_CURRENT_USER` (HKCU), and local machine, `HKEY_LOCAL_MACHINE` (HKLM), registry hives.

A new drive named `HKCC` might be created for `HKEY_CURRENT_CONFIG` with the following command:

```
New-PSDrive HKCC -PSProvider Registry -Root HKEY_CURRENT_CONFIG
```

After running the preceding command, a new drive may be used to view the content of the hive, as demonstrated by the following example:

```
PS C:\> Get-ChildItem HKCC:
    Hive: HKEY_CURRENT_CONFIG
Name                           Property
----                           --------
Software
System
```

Functions for drive letters

Running `C:` or `D:` in the PowerShell console changes to a new drive letter. This is possible because `C:` is a function that calls the `Set-Location` command. This can be seen by looking at the definition of one of the functions:

```
(Get-Command C:).Definition
```

Every letter of the alphabet (A to Z) has a predefined function (`Get-Command *:`).

`Set-Location` must be explicitly used to change to any other drive; for example:

```
Set-Location HKCU:
```

Introduction to splatting

Splatting is a way of defining the parameters of a command before calling it. This is an important and often underrated technique that the PowerShell team added in PowerShell 2.

Splatting is often used to solve three potential problems in a script:

- Long lines caused by commands that need many parameters
- Conditional use of parameters
- Repetition of parameters across several commands

Individual parameters are written in a hashtable (`@{}`), and then the `@` symbol is used to tell PowerShell that the content of the hashtable should be read as parameters.

This example supplies the `Name` parameter for the `Get-Process` command, and is normally written as `Get-Process -Name explorer` :

```
$getProcess = @{
    Name = 'explorer'
}
Get-Process @getProcess
```

In this example, `getProcess` is used as the name of the variable for the hashtable. The name is arbitrary; any variable name can be used.

Splatting can be used with cmdlets, functions, and scripts. Splatting can be used when the call operator is present; for example:

```
$getProcess = @{
    Name = 'explorer'
}
& 'Get-Process' @getProcess
```

The ability to use splatting with the call operator is useful if the command name itself is held in a variable.

The uses of splatting are explored in the following sections.

Splatting to avoid long lines

The benefit of splatting is most obvious when working with commands that expect a larger number of parameters.

This first example uses the module `ScheduledTasks` to create a basic task that runs once a day at midnight:

```
$taskAction = New-ScheduledTaskAction -Execute pwsh.exe -Argument 'Write-Host "hello world"'
$taskTrigger = New-ScheduledTaskTrigger -Daily -At '00:00:00'
Register-ScheduledTask -TaskName 'TaskName' -Action $taskAction -Trigger $taskTrigger -RunLevel
'Limited' -Description 'This line is too long to read'
```

Each of the commands is spread out over multiple lines; it is hard to see where one ends and the next begins.

Commands can also be spread out using a tick, the escape character in PowerShell, as shown here:

```
$taskAction = New-ScheduledTaskAction `
    -Execute pwsh.exe `
    -Argument 'Write-Host "hello world"'
$taskTrigger = New-ScheduledTaskTrigger `
    -Daily `
    -At '00:00:00'
Register-ScheduledTask `
    -TaskName 'TaskName' `
    -Action $taskAction `
    -Trigger $taskTrigger `
    -RunLevel 'Limited' `
    -Description 'This line is too long to read'
```

This approach is relatively common, but it is fragile. It is easy to miss a tick from the end-of-line, or to accidentally add a space after a tick character. Both break continuation, and the command executes but with an incomplete set of parameters; afterward, an error may be displayed, or a prompt may be shown, depending on the parameter (or parameters) it missed.

This problem is shown in the following screenshot, where a space character has been accidentally included after a tick following the `Daily` switch parameter:

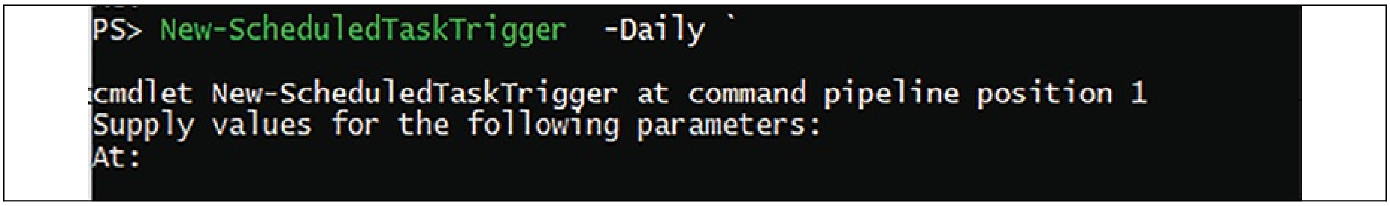


Figure 1.4: Space after the escape character

Splatting provides a neater, generally easier to read, and more robust alternative. The following example shows one possible way to tackle these commands when using splatting:

```
$newTaskAction = @{
    Execute = 'pwsh.exe'
    Argument = 'Write-Host "hello world"'
}
$newTaskTrigger = @{
    Daily = $true
    At    = '00:00:00'
}
$registerTask = @{
    TaskName    = 'TaskName'
    Action      = New-ScheduledTaskAction @newTaskAction
    Trigger     = New-ScheduledTaskTrigger @newTaskTrigger
    RunLevel    = 'Limited'
    Description = 'Splatting is easy to read'
}
Register-ScheduledTask @registerTask
```

Switch parameters may be treated as if they are Boolean when splatting.

The `Daily` parameter that was defined in the previous example is a switch parameter.

The same approach applies to `Confirm`, `Force`, `WhatIf`, `Verbose`, and so on.

Conditional use of parameters

Conditional use of parameters is one of the most important ways in which splatting can help.

If a command must be run and the parameters for a command can change based on user input or other circumstances, then it may be tempting to repeat the entire command. For example, a `Credential` parameter might be conditionally used.

The command may be repeated entirely based on there being a value for the `Credential` variable:

```
if ($Credential) {
    Get-ADUser 'Enabled -eq $true' -Credential $Credential
} else {
    Get-ADUser 'Enabled -eq $true'
}
```

The disadvantage of this approach is that any change to the command must be repeated in the second version.

Alternatively, a splatting variable may be used, and the `Credential` parameter added only when it is needed:

```
$params = @{}
if ($Credential) {
    $params['Credential'] = $Credential
}
Get-ADUser 'Enabled -eq $true' @params
```

Using splatting in this manner ensures only one version of the command must be maintained, reducing the risk of introducing a bug when making changes.

Splatting to avoid repetition

Splatting may be used to avoid repetition when a parameter must be optionally passed on to several different commands. It is possible to splat more than one set of parameters.

In this example, the `ComputerName` and `Credential` parameters are used by two different commands:

```
# Parameters used to authenticate remote connections
$remoteParams = @{
    Credential   = Get-Credential
    ComputerName = $env:COMPUTERNAME
}
# Parameters which are specific to Test-WSMan
$testWSMan = @{
    Authentication = 'Default'
    ErrorAction    = 'SilentlyContinue'
}
# By default, do not pass any extra parameters to New-CimSession
$newCimSession = @{}
if (-not (Test-WSMan @testWSMan @remoteParams)) {
    # If WSMan fails, use DCOM (RPC over TCP) to connect
    $newCimSession['SessionOption'] = New-CimSessionOption -Protocol Dcom
}
# Parameters to pass to Get-CimInstance
$getCimInstance = @{
    ClassName  = 'Win32_Service'
    CimSession = New-CimSession @newCimSession @remoteParams
}
Get-CimInstance @getCimInstance
```

This example takes advantage of several features:

- It is possible to splat no parameters using an empty hashtable (`@{}`) when all the parameters are conditionally added.
- It is possible to test conditions and dynamically add parameters at runtime (if needed).
- It is possible to splat more than one set of parameters into a command.

As the preceding example shows, it is possible to dynamically choose the parameters that are passed to a command without having to write the command in full more than once in a script.

Splatting and positional parameters

It is possible, although rare and inadvisable in production scripts, to splat positional parameters; that is, to splat a parameter without stating a parameter name.

This can be seen with the `Rename-Item` command, which has two positional parameters: `Path` and `NewName`. `Rename-Item` may be run as follows:

```
Rename-Item oldname.txt newname.txt
```

An array to splat these positional parameters looks as follows:

```
$renameItem = 'oldname.txt', 'newname.txt'
Rename-Item @renameItem
```

A splatting variable with positional parameters may be used with executable files (`.exe` files), although it is often difficult to see any difference between splatting and using a normal variable. For example, both of the following commands execute in the same way:

```
$argumentList = '/t', 2
timeout.exe $argumentList
timeout.exe @argumentList
```

Splatting is a powerful technique and can be used to make code more readable by reducing line length or repetition.

When using splatting, string values in the hashtable must be quoted. Conversely, when using a parameter directly, it is often unnecessary. The parser is responsible for deciding how statements and expressions should be interpreted.

Parser modes

The parser in PowerShell is responsible for taking what is typed into the console, or what is written in a script, and turning it into something PowerShell can execute. The parser has two different modes that explain, for instance, why strings assigned to variables must be quoted, but strings as arguments for parameters only need quoting if the string contains a space.

The parser modes are different modes:

- Argument mode
- Expression mode

Mode switching allows PowerShell to correctly interpret arguments without needing values to be quoted. In the following example, the argument for the `Name` parameter only needs quoting if the name contains spaces:

```
Get-Process -Name pwsh
```

The parser is running in Argument mode at the point the `pwsh` value is used and therefore literal text is treated as a value, not something to be executed.

This means that, in the following example, the second command is interpreted as a string and not executed:

```
Set-Content -Path commands.txt -Value 'Get-ChildItem', 'Get-Item'
Get-Command -Name Get-Content commands.txt
```

The second command in the preceding code therefore does not do anything.

To execute the `Get-Content` command, the argument must be enclosed in parentheses:

```
Set-Content -Path commands.txt -Value 'Get-ChildItem', 'Get-Item'
Get-Command -Name (Get-Content commands.txt)
```

The code in parentheses is executed, and the parser is in Expression mode.

Another example of this can be seen when using an enumeration value. An enumeration is a list of constants described by a .NET type. Enumerations are explored in *Chapter 7, Working with .NET*:

```
PS> Get-Date [DayOfWeek]::Monday
Get-Date: Cannot bind parameter 'Date'. Cannot convert value "[DayOfWeek]::Monday" to type
"System.DateTime". Error: "String '[DayOfWeek]::Monday' was not recognized as a valid DateTime."
```

If the value for the argument is placed in parentheses, it will run first and expand the value. Once the value is expanded, `Get-Date` will be able to work with it:

```
Get-Date ([DayOfWeek]::Monday)
```

The help document, `about_parsing`, explores this topic in greater detail.

Experimental features

PowerShell 7 uses experimental features to make some new functionality available, which is not yet considered to be a mainstream feature. Three commands are available for working with experimental features:

- `Enable-ExperimentalFeature`
- `Disable-ExperimentalFeature`
- `Get-ExperimentalFeature`

`Get-ExperimentalFeature` can be used to view the available features. The list of features changes, depending on the version of PowerShell. The following list has been taken from PowerShell 7.0.2.

```
PS> Get-ExperimentalFeature
Name                                Enabled Source
----                                ------- ------
PSCommandNotFoundSuggestion           False PSEngine
PSImplicitRemotingBatching            False PSEngine
PSNullConditionalOperators            False PSEngine
Microsoft.PowerShell.Utility.PSMan…   False C:\program files\powersh…
PSDesiredStateConfiguration.Invoke…   False C:\program files\powersh…
```

In addition to the output shown here, each feature has a description. `Format-Table` can be used to view these descriptions:

```
Get-ExperimentalFeature | Format-Table Name, Description -Wrap
```

The use of these commands is described in the `About_Experimental_Features` help file.

If an experimental feature is enabled, PowerShell must be restarted for us to use the feature. For example, `PSCommandNotFoundSuggestion` can be enabled:

```
Enable-ExperimentalFeature -Name PSCommandNotFoundSuggestion
```

Once enabled, if a command is spelt incorrectly in the console, PowerShell will suggest possible command names alongside the error message:

```
PS> Get-Procss
Get-Procss: The term 'Get-Procss' is not recognized as the name of a cmdlet, function, script file, or
operable program. Check the spelling of the name, or if a path was included, verify that the path is
correct and try again.
Suggestion [4,General]: The most similar commands are: Get-Process, Wait-Process, Get-Host, Get-
DbaProcess.
```

If the feature is no longer wanted, it can be disabled again:

```
Disable-ExperimentalFeature -Name PSCommandNotFoundSuggestion
```

In the past, experimental features have become permanent features in PowerShell.

Summary

This chapter contained several foundational topics for PowerShell, starting with picking an editor, using help content, and command discovery.

The ability to use the help system and discover commands is vital, regardless of skill level. The availability of help content in the shell allows new commands to be quickly incorporated and used.

Naming plays an important role in PowerShell. Strict use of a reasonably small set of verbs greatly enhances discovery and, reasonable assumptions can be made about a command before reaching for help content. PowerShell tends to use longer and descriptive command names compared with other scripting languages.

Once a command has been found, it is important to understand how to use the help content and the parameters a command offers to use it effectively.

Providers allow access to data in a similar manner to using a filesystem. Providers play and important role in PowerShell and are explored again later in this book when exploring the filesystem and registry.

Splatting was introduced and will be used repeatedly throughout this book. In the context of this book, it is primarily used to reduce line length. Splatting is an incredibly useful technique when scripting. The ability to conditionally use parameters without repeating code reduces complexity and the chance of introducing bugs.

The parser was introduced to explain command syntax, when values must be quoted, and when parentheses are required. The parser is complex and the examples in `about_parsing` should be reviewed.

Finally, PowerShell 6 introduced the idea of experimental features. This continues into PowerShell 7. You can toggle on (or off again) new features that have not become mainstream. Some of these features are explored again in *Chapter 4*, *Operators*.

Chapter 2, *Modules and Snap-Ins*, moves on to exploring modules and snap-ins, allowing PowerShell users to go beyond the base set of commands and include content published by others.

2

Modules and Snap-Ins

Modules are packaged collections of commands that may be loaded inside PowerShell, allowing PowerShell to interact with new systems and services. Modules come from a wide variety of different sources.

PowerShell itself is installed with a small number of modules, including `ThreadJob` and `PSReadline`.

You can install modules by adding Windows features or enabling capabilities, for example, the `ActiveDirectory` and `GroupPolicy` modules.

Some applications include modules; for example, Microsoft **Local Administrator Password Solution** (**LAPS**) includes a PowerShell module in the installer that you can use to manage some of the features of the application.

The Windows platform itself includes many modules, most of these having been included since Windows 8 was released.

Finally, you can install modules from the PowerShell Gallery or another registered repository. The PowerShell Gallery can include updated versions of PowerShell installed modules.

The PowerShell Gallery is therefore a valuable source of modules published by Microsoft, VMware, Amazon Web Services, and many others.

Snap-ins were included in PowerShell 1 and largely replaced with modules with the release of PowerShell 2. PowerShell 7 does not support snap-ins; snap-ins are limited to Windows PowerShell.

The chapter covers the following topics:

- Introducing modules
- Using Windows PowerShell modules in PowerShell 7
- PowerShellGet 3.0
- PowerShell repositories
- About snap-ins

Introducing modules

Modules were introduced with the release of PowerShell version 2.0. A module is a packaged set of commands that includes any required supporting content; modules often include help content.

Modules tend to target a specific system or focus on a small set of related operations. For example, the `Microsoft.PowerShell.Archive` module contains a small number of commands for interacting with ZIP files.

The modules available on a system can be discovered using the `Get-Module` command.

The Get-Module command

`Get-Module` is used to find the modules either in the current PowerShell session, or available on the current system.

PowerShell itself comes with several built-in modules, including `PowerShellGet`, `ThreadJob`, `PSReadLine`, and the commands in the `Microsoft.PowerShell.*` modules.

The Windows platform, especially the most recent versions, comes with a wide variety of modules installed. These, as well as any other available modules, can be viewed using the `Get-Module -ListAvailable` command.

By default, `Get-Module` returns information about each module that has been imported (either automatically or by using

`Import-Module`). For example, if the command is run from PowerShell 7, it shows that the `ISE` module has been loaded:

```
PS> Get-Module
ModuleType     Version     Name                             ExportedCommands
----------     -------     ----                             ----------------
Script         1.0.0.0     ISE                              {Get-IseSnippe...}
Manifest       3.1.0.0     Microsoft.PowerShell.Management {Add-Computer...}
```

The `ListAvailable` parameter shows the list of modules that are available on the system instead of just those that have been imported:

```
Get-Module -ListAvailable
```

Modules are discovered using the paths in the `PSModulePath` environment variable, which contains a delimited list of paths for PowerShell to search.

`Get-Module` will show all instances of a module regardless of the path and version when using the `All` parameter:

```
Get-Module <ModuleName> -All -ListAvailable
```

Modules that are available on a system can be imported either by running `Import-Module` or by running a command from the module.

The Import-Module command

PowerShell 3 and later attempts to automatically load modules if a command from that module is used and the module is under one of the paths in the `$env:PSModulePath` environment variable. Explicit use of the `Import-Module` command is less important than it was before Windows PowerShell 3.

For example, if PowerShell is started and the `CimCmdlets` module is not imported, running the `Get-CimInstance` command will cause the module to be automatically imported. This is shown in the following example:

```
PS> Get-Module CimCmdlets
PS> Get-CimInstance Win32_OperatingSystem | Out-Null
PS> Get-Module CimCmdlets
ModuleType Version    PreRelease Name        ExportedCommands
---------- -------    ---------- ----        ----------------
Binary     7.0.0.0               CimCmdlets  {Get-CimAssociatedInstance,...
```

The autoloader may be disabled using the `$PSModuleAutoLoadingPreference` variable as shown here:

```
$PSModuleAutoLoadingPreference = 'None'
```

You can explicitly import modules in PowerShell using the `Import-Module` command. Modules may be imported using a name or with a full path, as shown in the following example:

```
Import-Module -Name ThreadJob
Import-Module -Name $PSHome\Modules\ThreadJob\ThreadJob.psd1
```

Importing a module using a path is only required if the module is not in a discoverable path.

Once a module has been imported, the commands within the module may be listed using `Get-Command` as follows:

```
Get-Command -Module ThreadJob
```

> As the commands exported by a module are only identified by PowerShell importing the module, the previous command will also trigger an automatic import.

Modules installed in Windows PowerShell 5 and later are placed in a folder named after the module version, for example, `Modules\ModuleName\1.0.0\<ModuleContent>`. This allows multiple versions of the same module to coexist, as shown in the following example:

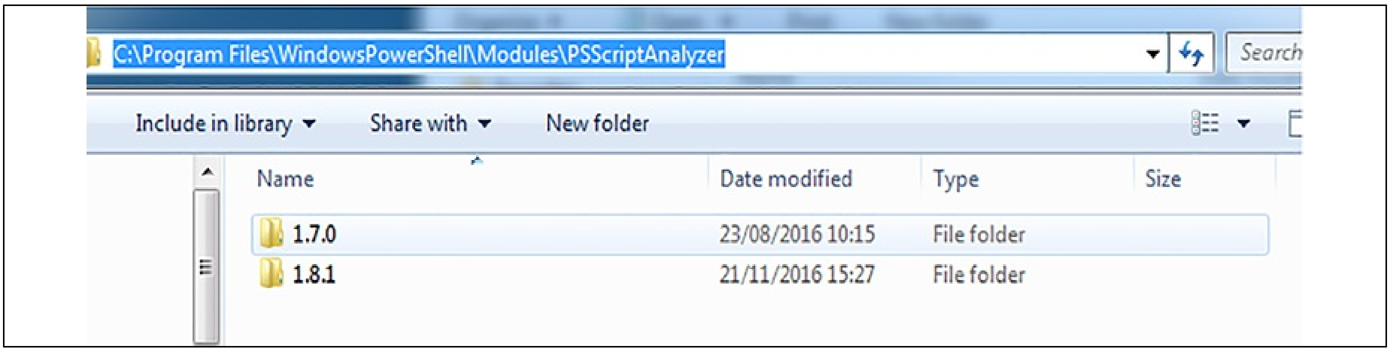


Figure 2.1: Side-by-side versioning

Version `1.8.1` of `PSScriptAnalyzer` will be imported by default, as it is the highest version number. It is possible to import a specific version of a module using the `MinimumVersion` and `MaximumVersion` parameters:

```
Import-Module PSScriptAnalyzer -MaxmimumVersion 1.7.0
```

Modules that have been imported can be removed from a PowerShell session using the `Remove-Module` command.

The Remove-Module command

The `Remove-Module` command removes a previously imported module from the current session.

For binary modules or manifest modules that incorporate a **Dynamic Link Library** (**DLL**), commands are removed from PowerShell but DLLs are not unloaded. DLL files used in a PowerShell session cannot be unloaded without restarting the PowerShell process.

`Remove-Module` does not remove or delete the files that make up a module from a computer.

Each of the preceding commands, by default, interacts with modules saved in the `PSModulePath` environment variable.

PSModulePath in PowerShell

`PSModulePath` is a delimited list of paths that can be used to store modules. You can import modules in these paths by name and they will be automatically loaded when a command from the module is used.

PowerShell allows the value of `$env:PSModulePath` to be set using **user**- and **machine**-scoped environment variables. By default, the **machine**-scoped variable should include the following paths, which are used by Windows PowerShell and by PowerShell 7 for compatibility:

```
C:\Program Files\WindowsPowerShell\Modules
C:\Windows\System32\WindowsPowerShell\v1.0\Modules
```

If the environment variables do not exist, PowerShell 7 uses the default values:

```
PS> $Env:PSModulePath -split ';'
C:\Users\whoami\Documents\PowerShell\Modules
C:\Program Files\PowerShell\Modules
c:\program files\powershell\7\Modules
```

The default values in the preceding list are included regardless of the value of the environment variable.

When using module paths, it is important to note that PowerShell does not search all paths for the latest version of a module. PowerShell searches the list of paths in the order they appear in the `PSModulePath` environment variable. If a module is listed in more than one path, the most recent version from the first discovered path is used.

For example, if the current user path contains a module with version 1.0.0, and the program files path contains the same module but with version 2.0.0, PowerShell will prefer to load version 1.0.0 because the current user path is searched first. The `Version` or `MinimumVersion` parameter must be used with `Import-Module` to avoid this.

If both Windows PowerShell and PowerShell 7 are in use in an environment, care must be taken when updating the `PSModulePath` environment variable. The behavior described previously differs from Windows PowerShell. In Windows PowerShell:

- If the *user* environment variable is set, it completely replaces the user value, which defaults to `C:\Users\whoami\Documents\WindowsPowerShell\Modules`
- If the *machine* environment variable is set, it replaces the `system32` path: `C:\windows\system32\windowspowershell\v1.0\Modules`
- In all cases, the `C:\Program Files\WindowsPowerShell\Modules` path remains

The `C:\windows\system32\windowspowershell\v1.0\Modules` path should be included in the machine environment variable to allow PowerShell 7 to load modules, either directly or using a Windows PowerShell compatibility session.

The value of `$env:PSModulePath` may be safely modified within on all PowerShell versions and all platforms, for example, by using a profile script. Changes made to `$env:PSModulePath` are scoped to the process and only affect the current PowerShell session and any child processes; the changes do not persist.

PowerShell 7 can use modules intended for Windows PowerShell either directly or by using a Windows PowerShell compatibility session.

Using Windows PowerShell modules in PowerShell 7

Many modules available to Windows PowerShell are compatible with PowerShell 7 without requiring any changes.

If a module is not compatible with PowerShell 7, an attempt can be made to load the module in a Windows compatibility session.

In PowerShell 6, the following functionality discussed is part of the `WindowsCompatibility` module available in the PowerShell Gallery. This module is not required in PowerShell 7. In PowerShell 7, the ability to load a module in a compatibility session is built into the `Import-Module` command.

The `TLS` module, for example, will not load PowerShell 7 by default because it does not state that it supports the Core edition of PowerShell, as shown by `Get-Module`:

```
PS> Get-Module TLS -ListAvailable -SkipEditionCheck
    Directory: C:\Windows\System32\WindowsPowerShell\v1.0\Modules
ModuleType Version    PreRelease Name       PSEdition ExportedCommands
---------- -------    ---------- ----       --------- ----------------
Manifest   2.0.0.0               TLS        Desk      {New-TlsSessionTic...
```

The module can be loaded in two ways:

The edition check can be skipped (the module may work, it may just lack testing, and therefore careful testing may be required before using the module in a production scenario):

```
Import-Module TLS -SkipEditionCheck
```

Alternatively, if the previous command fails, the module may load in a compatibility session:

```
PS> Import-Module TLS -UseWindowsPowerShell
WARNING: Module TLS is loaded in Windows PowerShell using WinPSCompatSession remoting session; please
note that all input and output of commands from this module will be deserialized objects. If you want
to load this module into PowerShell Core please use 'Import-Module -SkipEditionCheck' syntax.
```

The compatibility session can be seen using the `Get-PSSession` command after the module has been imported:

```
Get-PSSession -Name WinPSCompatSession
```

When importing the preceding `TLS` module, a warning message is displayed that notes the input and output is deserialized. The impact of this depends on the complexity of the objects returned by the command; typically, it will mean that methods of specialized types will not work from PowerShell 7.

The effect of this can be demonstrated by invoking `Get-WmiObject` in the compatibility session. `Get-WmiObject` is not available in PowerShell 7 and cannot be directly used:

```
$session = Get-PSSession -Name WinPSCompatSession
$process = Invoke-Command -Session $session -ScriptBlock {
    Get-WmiObject Win32_Process -Filter "ProcessID=$PID"
}
```

If the `Get-WmiObject` command is run in Windows PowerShell without using `Invoke-Command`, it will have several methods available. One of these methods is `GetRelated`, which is typically used as follows when used in Windows PowerShell:

```
$process = Get-WmiObject Win32_Process -Filter "ProcessID=$PID"
$process.GetRelated('Win32_Session')
```

Because PowerShell 7 has a copy of the properties only, the method does not exist and an error will be displayed:

```
PS> $session = Get-PSSession -Name WinPSCompatSession
PS> $process = Invoke-Command -Session $session -ScriptBlock {
>>     Get-WmiObject Win32_Process -Filter "ProcessID=$PID"
>> }
PS> $process.GetRelated('Win32_Session')
InvalidOperation: Method invocation failed because
[Deserialized.System.Management.ManagementObject#root\cimv2\Win32_Process] does not contain a method
named 'GetRelated'.
```

The compatibility feature is incredibly useful but does not replace native compatibility with modern versions of PowerShell.

PowerShell on the Windows platform has a wide variety of modules available, or available through installable applications and features to interact with other systems. New modules can also be installed from resources such as the PowerShell Gallery.

Finding and installing modules

PowerShell includes a module named `PowerShellGet`, which can be used to register repositories and search for and install modules.

By default, `PowerShellGet` searches the PowerShell Gallery.

What is the PowerShell Gallery?

The PowerShell Gallery is a Microsoft-run repository and distribution platform for PowerShell scripts and modules written by Microsoft or other users.

The PowerShell Gallery has parallels in other scripting languages, as shown in the following examples:

* Perl has cpan.org
* Python has PyPI
* Ruby has RubyGems

Support for the gallery is included by default in PowerShell 5 and above. For Windows PowerShell 3 and 4, PowerShellGet must be installed as described in Microsoft Docs: https://docs.microsoft.com/powershell/scripting/gallery/installing-psget.

The PowerShell Gallery may be searched using https://www.powershellgallery.com as shown in the following screenshot:

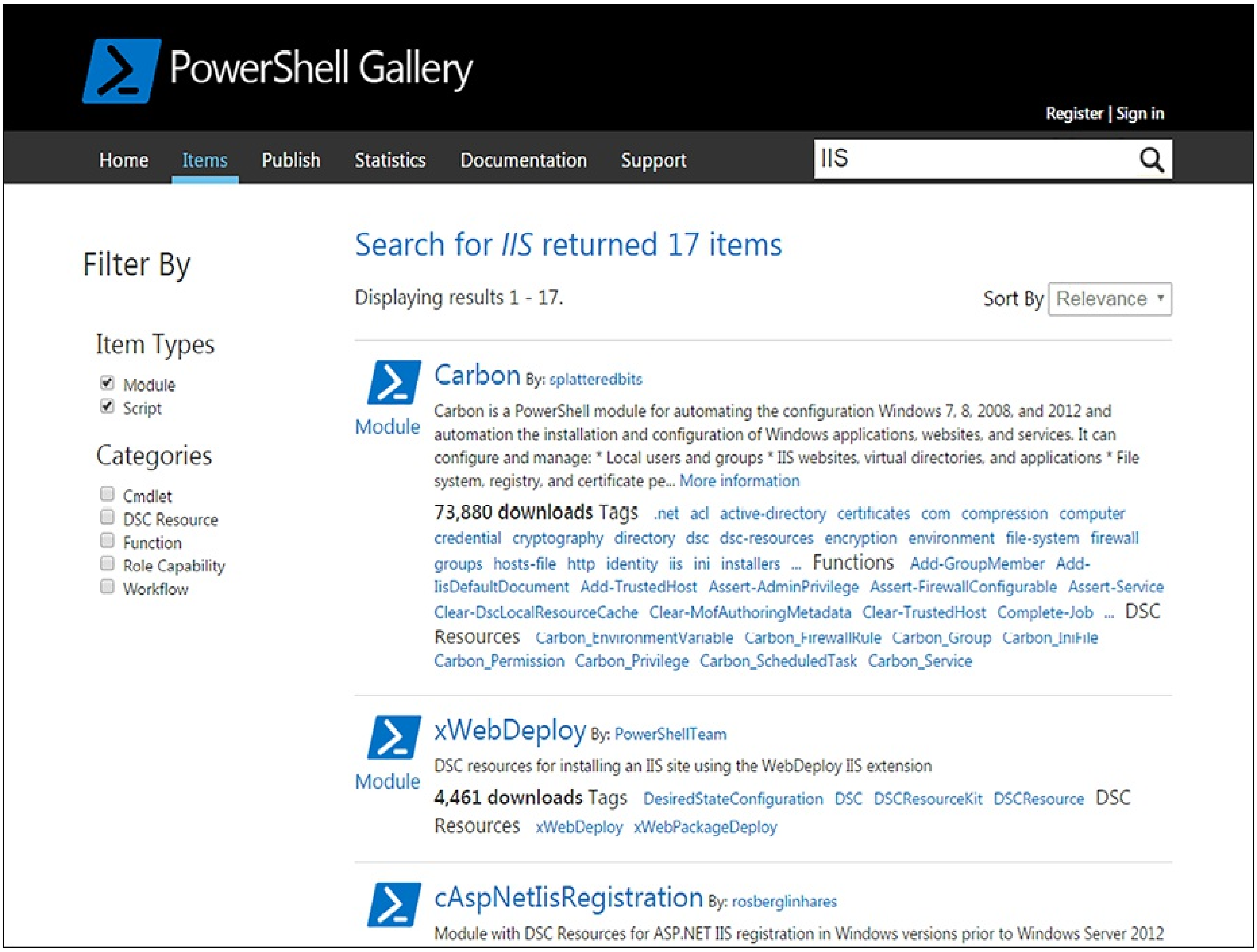


Figure 2.2: Searching the PowerShell Gallery

You can use the `Find-Module` command to search the PowerShell Gallery, or any registered repository, instead of using the web page.

The Find-Module command

`Find-Module` is used to search registered PowerShell repositories. Modules are identified by name, as shown in the following example:

```
Find-Module Carbon
Find-Module -Name Carbon
Find-Module -Name Azure*
```

You can use the `Filter` parameter when the name alone is not sufficient to find an appropriate module. Supplying a value for the `Filter` parameter is equivalent to using the search field in the PowerShell Gallery, it expands the search to include tags:

```
Find-Module -Filter IIS
```

The `Find-Module` command cannot filter based on PowerShell edition, and the result of the search does not state which version the module might work with.

Once found, a module can be installed using the `Install-Module` command.

The Install-Module command

The `Install-Module` command installs modules from the PowerShell Gallery or any other configured repository. By default, `Install-Module` adds modules to the path for `AllUsers`, at `C:\Program Files\PowerShell\Modules` on Windows and at `/usr/local/share/powershell/Modules` on Ubuntu.

> **Access rights**
>
> Installing a module under the `AllUsers` scope requires administrative access.

For example, the `posh-git` module may be installed using either of the following two commands:

```
Find-Module -Name posh-git | Install-Module
Install-Module posh-git
```

Modules may be installed under a user-specific path (`$home\Documents\WindowsPowerShell\Modules`) using the `Scope` parameter:

```
Install-Module carbon -Scope CurrentUser
```

If the most recent version of a module is already installed, the command ends without providing feedback. If a newer version is available, it will be automatically installed alongside the original.

The `Force` parameter may be used to reinstall a module:

```
Install-Module posh-git -Force
```

`Force` may also be used to install a newer version of a module when the existing version was not installed from a PS repository, or when changing the scope a module is installed in.

The `Install-Module` command does not provide an option to install modules under the `$PSHOME` directory. The `$PSHOME` directory is reserved for modules that are shipped with the PowerShell installer, or for Windows PowerShell, those that are shipped with the Windows operating system.

The Update-Module command

You can use the `Update-Module` command to update any module installed using the `Install-Module` command.

In both Windows PowerShell and PowerShell 7, `Update-Module` attempts to update the specified module to the latest or specified version.

The Save-Module command

The `Save-Module` command downloads the module from the PowerShell Gallery (or any other registered repository) to a path without installing it. This is useful if you save a module to an alternate module path, or if you intend to copy the module onto a system that cannot use `Install-Module`.

The following example command downloads the `Carbon` module into a `Modules` directory in the root of the `C:` drive:

```
Save-Module -Name Carbon -Path C:\Modules
```

`Save-Module` will download the module and overwrite any previously saved version in the specified path. The command ignores other downloaded versions of the module.

Each of the preceding commands is part of `PowerShellGet` 2. `PowerShellGet` 3 is likely to be released in the coming months and adopts a slightly different approach to the commands.

PowerShellGet 3.0

`PowerShellGet` 2 (for example, `PowerShellGet` 2.2.4.1) implements the `Install-Module`, `Update-Module`, and `Save-Module` module commands demonstrated at the beginning of this chapter.

`PowerShellGet` 3.0 is in preview at the time of writing; the following commands are based on the beta7 pre-release. One of the key features is that this new version does not depend on the `PackageManagement` module, allowing a simpler installation process, avoiding the need to bootstrap the NuGet provider, making upgrading the module simpler.

The preview version also uses new command names, completely divorcing it from the previous implementations of `PowerShellGet`. The change in command names means the new version can safely be installed alongside any existing version.

`PowerShellGet` 3.0 can be installed as follows:

```
Install-Module PowerShellGet -Force -AllowPrerelease
```

Once installed, you'll need to register the PowerShell Gallery or another repository:

```
Register-PSResourceRepository -PSGallery
```

In `PowerShellGet` 2.0, you implement separate commands to work with modules and scripts. `PowerShellGet` 3.0 does not differentiate between modules and scripts; all artifacts are termed `PSResource`, and all searches use the `Find-PSResource` command. For example, we can find a module by using the following command:

```
Find-PSResource -Name Indented.Net.IP -Type Module
```

The `Type` parameter may be omitted without affecting the search results in this case.

Most of the commands in `PowerShellGet` 3.0 use the same approach as those in `PowerShellGet` 2.2.4 and below. Over time, the differences between the commands are likely to start to appear; for example, `Install-PSResource` includes a `Reinstall` parameter, which is somewhat like the `Force` parameter for `Install-Module` in `PowerShellGet` 2.

Repositories

Like older versions of `PowerShellGet`, repositories are registered on a per-user basis. In `PowerShellGet` 2.2.4 and below, the repository configuration file is found in the following path:

```
$env:LOCALAPPDATA\Microsoft\Windows\PowerShell\PowerShellGet\PSRepositories.xml
```

The `PSRepositories.xml` file is stored in CliXml format and may be read using the `Import-CliXml` command. The file is normally read and updated using `Get-PSRepository`, `Register-PSRepository`, and `Unregister-PSRespository`.

`PowerShellGet` 3.0 uses a simpler format for the `PSResourceRespository.xml` file. The file may be found in the following path:

```
$env:LOCALAPPDATA\PowerShellGet\PSResourceRepository.xml
```

The `Get-PSResourceRepository`, `Register-PSResourceRepository`, and `Unregister-PSResourceRepository` commands are the expected way of interacting with this file.

As with older versions of `PowerShellGet`, storing credentials for a repository is not currently supported. If a repository requires authentication, the `Credential` parameter must be used explicitly with each operation.

Version ranges

`Find-PSResource` allows wildcards to be used for the `Version` parameter; using `*` will return all available versions except pre-release. The `Prerelease` parameter may be added to include those:

```
Find-PSResource -Name PowerShellGet -Version *
```

A range of versions may be defined using the range syntax used by NuGet, which is described in the following document:

https://docs.microsoft.com/nuget/concepts/package-versioning#version-ranges-and-wildcards

For example, the highest version of `PowerShellGet` available between 1.0 (inclusive) and 2.0 (exclusive) may be found using this:

```
Find-PSResource -Name PowerShellGet -Version '[1.0,2.0)'
```

The search can be changed to be inclusive by changing the closing `)` to `]`. For example, the following command will find version 2.0.0 of `PowerShellGet`:

```
Find-PSResource -Name PowerShellGet -Version '[1.0,2.0]'
```

The same syntax will be available when declaring dependencies between modules.

PowerShell repositories

Each of the examples from the previous section uses the PowerShell Gallery as a source for installing modules. This is an important resource, but in a business setting, it may be desirable to restrict access to the gallery. Instead, an internal repository that holds curated or internally developed content may be implemented to share content.

Creating an SMB repository

A file share is a simple way to share content. Such a repository requires little effort to set up and maintain. A file share may be registered as a repository as follows:

```
$params = @{
    Name               = 'Internal'
    SourceLocation     = '\\server\share\directory'
    InstallationPolicy = 'Trusted'
}
Register-PSRepository @params
```

Existing modules can be published to the repository using the `Publish-Module` command. For example, if Pester 5.0.2 is installed, it may be published to the newly created internal repository:

```
$params = @{
    Name            = 'pester'
    RequiredVersion = '5.0.2'
    Repository      = 'Internal'
}
Publish-Module @params
```

The `RequiredVersion` parameter is mandatory if more than one version of the module (in this case Pester) exists on the system publishing the module. Once published, a `nupkg` file will appear in the file share. The Pester module is now available for installation by anyone else with the repository registered.

Users installing content from an SMB share must be authenticated and must have at least read access to the share. Guest access may be granted to avoid the authentication requirement.

NuGet repositories

NuGet is a package manager for .NET. `PowerShellGet` can use a NuGet repository as a source for PowerShell modules. The PowerShell Gallery is a NuGet repository.

NuGet offers greater flexibility when dealing with authentication, or package life cycles, when compared with SMB shares.

At the simple end, the **Chocolatey.Server package** available from chocolatey.org may be used to configure an **Internet Information Services** (**IIS**) website to act as a NuGet repository:

https://chocolatey.org/packages/chocolatey.server

> **About Chocolatey**
>
> Chocolatey is a package manager for Windows. See https://chocolatey.org for further information.

More advanced servers include Sonatype Nexus and ProGet. Both offer free to use servers, which may be locally deployed. These services must be configured, and once configured, packages will typically be published by using an API key to authenticate.

About snap-ins

Snap-ins, and the commands for interacting with snap-ins, are only available in Windows PowerShell; they are not present in PowerShell 7.

A snap-in is the precursor to a module. It was the mechanism available to extend the set of commands in PowerShell 1 and was deprecated with the release of PowerShell 2.

You can view the list of installed snap-ins by using the following command:

```
Get-PSSnapIn -Registered
```

If the `Registered` parameter is excluded, `Get-PSSnapIn` will show the snap-ins that have been imported into the current PowerShell session.

PowerShell does not automatically load commands from a snap-in. All snap-ins must be explicitly imported using the `Add-PSSnapIn` command:

```
Add-PSSnapIn WDeploySnapin3.0
```

Once a snap-in has been installed (registered) and added, you can use `Get-Command` to list the commands as if the snap-in was a module:

```
Get-Command -Module WDeploySnapin3.0
```

The snap-in shown will only be visible if the Web Deployment Toolkit 3.0 is installed.

Summary

Modules are a vital part of PowerShell. Modules allow users to extend the commands available within PowerShell, allowing PowerShell to work with many different systems from many different vendors.

The commands explored in this chapter have demonstrated how to discover and use locally available modules along with the commands each module contains. The PowerShell Gallery has been introduced as a public repository of modules, extending PowerShell further still.

PowerShellGet has been a feature of PowerShell since PowerShell 3. With the release of PowerShellGet 3 on the horizon, we demonstrated its new commands and filtering capabilities.

SMB- and NuGet-based repositories were briefly introduced for those looking to establish private repositories for use within an organization. This allows administrators to create private repositories with curated content, reducing exposure to the unknown modules.

Snap-ins, an artifact of PowerShell 1 that is limited to Windows PowerShell, were very briefly demonstrated for the products where snap-ins remain important.

Chapter 3, *Working with Objects in PowerShell*, dives into the commands available to work with objects in PowerShell, including `Where-Object` and `ForEach-Object`.

3

Working with Objects in PowerShell

Everything we do in PowerShell revolves around working with objects. Objects, in PowerShell, may have properties or methods (or both). It is difficult to describe an object without resorting to this; an object is a representation of a thing or item of data. Let's use an analogy to attempt to give meaning to the term.

A book is an object and has properties that describe its physical characteristics, such as the number of pages, the weight, or size. It has metadata (information about data) properties that describe the author, the publisher, the table of contents, and so on.

The book might also have methods. A method affects the state of an object. For example, there might be methods to open or close the book or methods to jump to different chapters. A method might also convert an object into a different format. For example, there might be a method to copy a page, or even destructive methods such as one to split the book.

PowerShell has a variety of commands that work with arrays (or collections) of objects in a pipeline.

In this chapter, we are going to cover the following topics:

- Pipelines
- Members
- Enumerating and filtering
- Selecting and sorting
- Grouping and measuring
- Comparing
- Importing, exporting, and converting

Pipelines

The pipeline is one of the most prominent features of PowerShell. The pipeline is used to send output from one command to another command as input.

Most of the output from a command is sent to what is known as **standard output**, often shortened to **stdout**.

Standard output

The term standard output is used because there are different kinds of output. Each of these different types of output is sent to a different stream, allowing each to be read separately. In PowerShell, the streams are **Standard**, **Error**, **Warning**, **Verbose**, **Debug**, and **Information**.

When assigning the output of a command to a variable, the assigned value is taken from the standard output (the output stream) of a command. For example, the following command assigns the data from the standard output to a variable:

```
$computerSystem = Get-CimInstance -ClassName Win32_ComputerSystem
```

Non-standard output, such as Verbose, will not be assigned to the variable.

Non-standard output

In PowerShell, each of the streams has a command associated with it:

Stream	**Command**	**Stream number**
Standard output	`Write-Output`	1
Error	`Write-Error`	2
Warning	`Write-Warning`	3
Verbose	`Write-Verbose`	4
Debug	`Write-Debug`	5
Information	`Write-Information`	6

Table 3.1: Streams and their associated commands

In PowerShell 5 and later, the `Write-Host` command is a wrapper for `Write-Information`. It sends output to the information stream.

Prior to Windows PowerShell 5, `Write-Host` did not have a dedicated stream; the output could only be captured via a transcript, that is, by using the `Start-Transcript` command to log console output to a file.

For example, if you add the `Verbose` switch to the preceding command, more information is shown. This extra information is not held in the variable; it is sent to a different stream:

```
PS> $computerSystem = Get-CimInstance Win32_ComputerSystem -Verbose
VERBOSE: Perform operation 'Enumerate CimInstances' with following parameters, ''namespaceName' =
root\cimv2,'className' = Win32_ComputerSystem'.
VERBOSE: Operation 'Enumerate CimInstances' complete.
PS> $computerSystem
Name    PrimaryOwnerName    Domain       TotalPhysicalMemory    Model
----    ----------------    ------       -------------------    -----
NAME    Username            WORKGROUP    17076875264            Model
```

The object pipeline

Languages such as Batch scripting (on Windows) or tools on Linux and Unix often use a pipeline to pass text between commands. When the output from one command is piped to another, it is up to the next command in the pipeline to figure out what the text from the input pipeline means.

PowerShell, on the other hand, sends objects from one command to another when using the pipeline.

The pipe (`|`) symbol is used to send the standard output between commands.

In the following example, the output of `Get-Process` is sent to the `Where-Object` command, which applies a filter. The filter restricts the list of processes to those that are using more than `50MB` of memory:

```
Get-Process | Where-Object WorkingSet64 -gt 50MB
```

When piping commands, a line break may be added after a pipe:

```
Get-Process |
    Where-Object WorkingSet64 -gt 50MB |
    Select-Object -Property Name, ID
```

Or, in PowerShell 7, the pipe may be placed at the start of the following line within a script or script block. To demonstrate in the console, hold Shift and press Return at the end of each line; the prompt will change to `>>` for each subsequent line:

```
Get-Process
    | Where-Object WorkingSet -gt 50MB
    | Select-Object Name, ID
```

When the pipe is placed at the end of a line, the command can be pasted or typed in the console as-is. When using a pipe at the beginning of a new line, Shift + Return must be used in the console to add the new line without ending the code block.

No special action is required in a script or function to use the pipe at the beginning of the line.

Members

At the beginning of this chapter, the idea of properties and methods was introduced. These are part of a set of items collectively known as members. These members are used to interact with an object. A few of the more frequently used members are `NoteProperty`, `ScriptProperty`, `ScriptMethod`, and `Event`.

> **What are the member types?**
>
> The list of possible member types can be viewed on MSDN, which includes a short description of each member type:
>
> https://docs.microsoft.com/dotnet/api/system.management.automation.psmembertypes?redirectedfrom=MSDN&view=pscore-6.2.0

This chapter focuses on the property members `Property`, `NoteProperty`, and `ScriptProperty`.

The Get-Member command

The `Get-Member` command can be used to view the different members of an object. For example, it can be used to list the members of a process object returned by `Get-Process`. The automatic variable `$PID` holds the process ID of the current PowerShell process:

```
Get-Process -Id $PID | Get-Member
```

`Get-Member` offers filters using its parameters (`MemberType`, `Static`, and `View`). For example, to view the properties of the PowerShell process object, the following can be used:

```
Get-Process -Id $PID | Get-Member -MemberType Property
```

The `Static` parameter is covered in *Chapter 7, Working with .NET*.

The `View` parameter is set to `All` by default. It has three additional values:

- **Base**: This shows properties that are derived from a .NET object
- **Adapted**: This shows members handled by PowerShell's **Adapted Type System** (**ATS**)
- **Extended**: This shows members added by PowerShell's **Extended Type System** (**ETS**)

> **ATS and ETS**
>
> ATS and ETS make it easy to work with object frameworks other than .NET in PowerShell, for example, objects returned by ADSI, COM, WMI, or XML. Each of these frameworks is discussed later in this book.
>
> Microsoft published an article on ATS and ETS in 2011, which is still relevant today:
>
> https://docs.microsoft.com/en-us/archive/blogs/besidethepoint/psobject-and-the-adapted-and-extended-type-systems-ats-and-ets

Accessing object properties

The properties of an object in PowerShell may be accessed by writing the property name after a period. For example, the `Name` property of the current PowerShell process may be accessed by using the following code:

```
$process = Get-Process -Id $PID
$process.Name
```

PowerShell also allows us to access these properties by enclosing a command in parentheses:

```
(Get-Process -Id $PID).Name
```

The properties of an object are objects themselves. For example, the `StartTime` property of a process is a `DateTime` object. The `DayOfWeek` property may be accessed using the following code:

```
$process = Get-Process -Id $PID
$process.StartTime.DayOfWeek
```

The variable assignment step may be skipped if parentheses are used:

```
(Get-Process -Id $PID).StartTime.DayOfWeek
```

If a property name has a space, it may be accessed using a number of different notation styles. For example, a property named `'Some Name'` may be accessed by quoting the name or enclosing the name in curly braces:

```
$object = [PSCustomObject]@{ 'Some Name' = 'Value' }
$object."Some Name"
$object.'Some Name'
$object.{Some Name}
```

A variable may also be used to describe a property name:

```
PS> $object = [PSCustomObject]@{ 'Some Name' = 'Value' }
PS> $propertyName = 'Some Name'
PS> $object.$propertyName
Value
```

The ability to assign a property or assign a new value to a property is governed by the access modifiers for a property.

Access modifiers

An access modifier for a property has two possible values: `Get`, indicating that the property can be read; and `Set`, indicating that the property can be written to (changed).

Depending on the type of object, properties may be read-only or read/write. These may be identified using `Get-Member` and by inspecting the access modifiers.

In the following example, the values in curly braces at the end of each line is the access modifier:

```
PS> $File = New-Item NewFile.txt
PS> $File | Get-Member -MemberType Property
    TypeName: System.IO.FileInfo
Name                MemberType    Definition
----                ----------    ----------
Attributes          Property      System.IO.FileAttributes Attributes {get;set;}
CreationTime        Property      datetime CreationTime {get;set;}
CreationTimeUtc     Property      datetime CreationTimeUtc {get;set;}
Directory           Property      System.IO.DirectoryInfo Directory {get;}
DirectoryName       Property      string DirectoryName {get;}
Exists              Property      bool Exists {get;}
```

When the modifier is `{get;}`, the property value is read-only; attempting to change the value results in an error:

```
PS> $File = New-Item NewFile.txt -Force
PS> $File.Name = 'NewName'
InvalidOperation: 'Name' is a ReadOnly property.
```

When the modifier is `{get;set;}`, the property value may be read and changed. In the preceding example, `CreationTime` has the set access modifier. The value can be changed; in this case, it may be set to any date after January 1, 1601:

```
$File = New-Item NewFile.txt -Force
$File.CreationTime = Get-Date -Day 1 -Month 2 -Year 1692
```

The result of the preceding command can be seen by reviewing the properties for the file in PowerShell:

```
Get-Item NewFile.txt | Select-Object -ExpandProperty CreationTime
```

Alternatively, you can use File Explorer to view a file's properties, as shown in *Figure 3.1*:

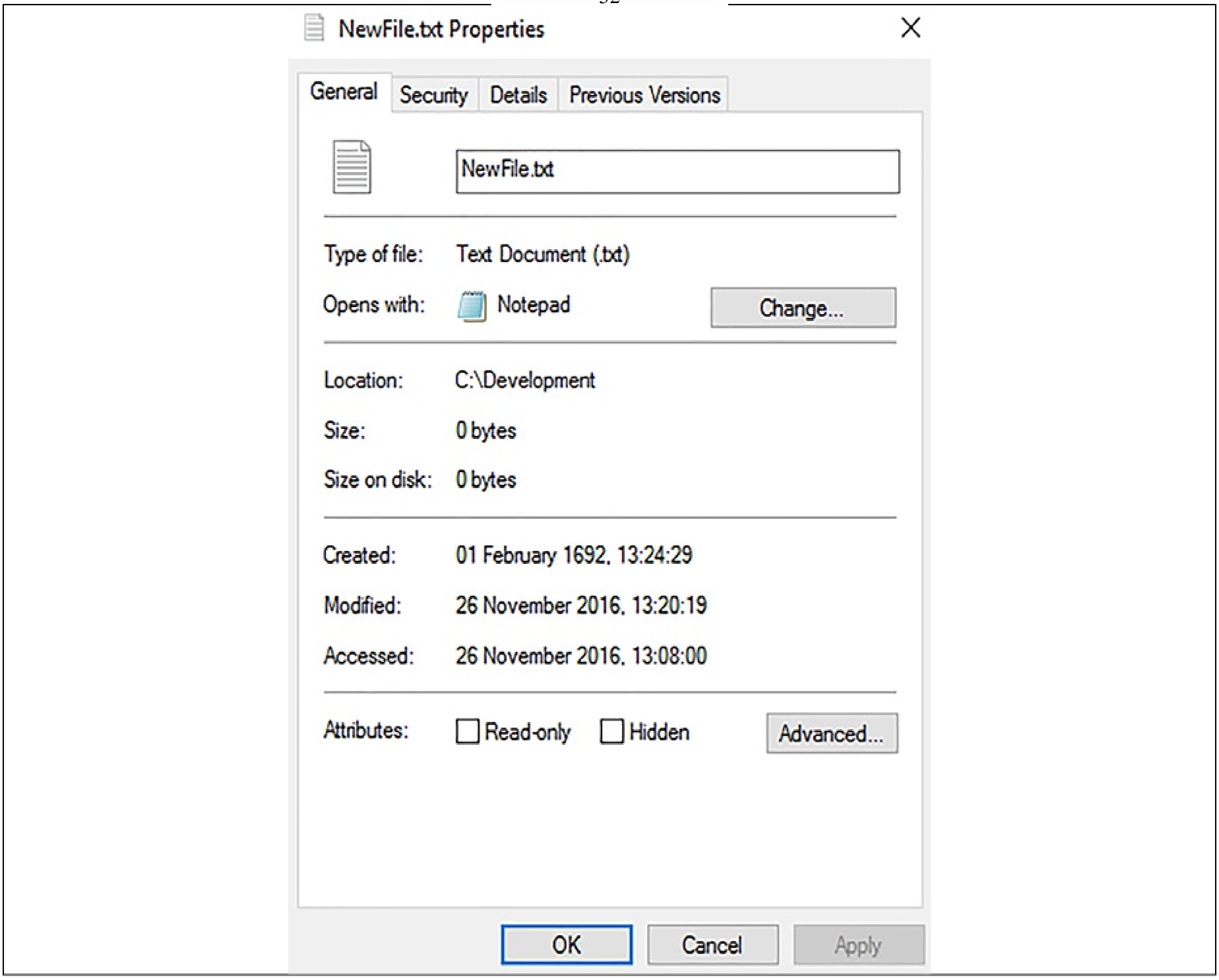


Figure 3.1: Changing the Created date

In the preceding example, the change made to `CreationTime` is passed from the object representing the file to the file itself. The object used here, based on the .NET class `System.IO.FileInfo`, is written in such a way that it supports the change. A property may indicate that it can be changed (by supporting the set access modifier in `Get-Member`) and still not pass the change back to whatever the object represents.

Using methods

Methods effect a change in state. That may be a change to the object associated with the method, or it may take the object and convert it into something else.

Methods are called using the following notation in PowerShell:

```
<Object>.Method()
```

When a method is called without parentheses, PowerShell will show the overload definitions. Overloading a method is a .NET concept; it comes into play when two or more methods share the same name but have different arguments and implementations:

```
PS> 'thisString'.Substring
OverloadDefinitions
-------------------
string Substring(int startIndex)
string Substring(int startIndex, int length)
```

An example of a method that takes an object and converts it into something else is shown here. In this case, a date is converted into a string:

```
PS> $date = Get-Date -Day 1 -Month 1 -Year 2010
PS> $date.ToLongDateString()
01 January 2010
```

The string generated in the preceding command is dependent on the current culture in use by PowerShell. In the preceding example, the culture is set to en-GB (Great Britain).

If a method expects to have arguments (or parameters), the notation becomes the following:

```
<Object>.Method(Argument1, Argument2)
```

An example of a method that takes arguments might be the `ToString` method on a `DateTime` object. `Get-Date` can be used to create a `DateTime` object using the current date:

```
PS> (Get-Date).ToString('u')
2016-12-08 21:18:49Z
```

In the preceding example, the letter `u` is one of the standard date and time format strings (https://docs.microsoft.com/dotnet/standard/base-types/standard-date-and-time-format-strings) and represents a universal sortable date/time pattern. The same result may be achieved by using the `Format` parameter of `Get-Date`:

```
PS> Get-Date -Format u
2016-12-08 21:19:31Z
```

The advantage that this method on a `DateTime` object has over the parameter for `Get-Date` is that the date can be adjusted before conversion by using some of the other properties and methods:

```
(Get-Date).Date.AddDays(-1).ToString('u')
```

The result of this command is the start of yesterday (midnight, one day before today).

An example of a method that changes a state might be seen on a `tcpClient` object. TCP connections must be opened before data can be sent over a network. The following example creates a `tcpClient` object, then opens a connection to the RPC Endpoint Mapper on the current computer:

```
$tcpClient = New-Object System.Net.Sockets.TcpClient
$tcpClient.Connect("127.0.0.1", 135)
```

The `Connect` method does not return anything, although it will throw an error if the connection fails. The method affects the state of the object and is reflected by the `Connected` property:

```
PS> $tcpClient.Connected
True
```

The state of the object may be changed again by calling the `Close` method to disconnect:

```
$tcpClient.Close()
```

The Add-Member command

`Add-Member` allows new members to be added to existing objects. This can be useful if the object type must be preserved.

`Select-Object` can also add new members, but it changes the object type when doing so.

Starting with an empty object, it is possible to add new properties:

```
PS> $empty = New-Object Object
PS> $empty |
>>     Add-Member -NotePropertyName New -NotePropertyValue 'Hello world'
PS> $empty
New
---
Hello world
```

`Add-Member` may also add a `ScriptProperty` or a `ScriptMethod` member. When writing script-based properties and methods, the reserved variable `$this` is used to refer to itself:

```
PS> $empty = New-Object Object
PS> $empty |
>>     Add-Member -NotePropertyName New -NotePropertyValue 'Hello world'
PS> $empty |
>>     Add-Member -Name Calculated -MemberType ScriptProperty -Value {
>>         $this.New.Length
>>     }
PS> $empty
New            Calculated
---            ----------
Hello world            11
```

Methods may be added as well, for example, a method to replace the word `world` in the new property:

```
$empty = New-Object Object
$empty | Add-Member -NotePropertyName New -NotePropertyValue 'Hello world'
$params = @{
    Name       = 'Replace'
    MemberType = 'ScriptMethod'
    Value      = { $this.New -replace 'world', 'everyone' }
}
$empty | Add-Member @params
```

Running the newly added `Replace` method will show the updated string:

```
PS> $empty.Replace()
Hello everyone
```

As well as being used to present information to an end user, the properties and methods of an object are used when enumerating and filtering collections of objects.

Enumerating and filtering

Enumerating, or listing, the objects in a collection in PowerShell does not need a specialized command. For example, if the results of `Get-PSDrive` were assigned to a variable, enumerating the content of the variable is as simple as writing the variable name and pressing Return, allowing the values to be viewed:

```
PS> $drives = Get-PSDrive
PS> $drives
Name     Used (GB)    Free (GB)    Provider      Root
----     ---------    ---------    --------      ----
Alias                              Alias
C           319.37       611.60    FileSystem    C:\
Cert                               Certificate   \
Env                                Environment
...
```

`ForEach-Object` may be used to work on an existing collection or objects, or used to work on the output from another command in a pipeline.

`Where-Object` may be used to filter an existing collection or objects, or it may be used to filter the output from another command in a pipeline.

The ForEach-Object command

`ForEach-Object` is used to work on each object in an input pipeline. For example, the following command works on each of the results from `Get-Process` in turn by running the specified script block (enclosed in `{ }`):

```
Get-Process | ForEach-Object {
    Write-Host $_.Name -ForegroundColor Green
}
```

The script block, an arbitrary block of code, is used as an argument for the `Process` parameter. The preceding command may explicitly include the `Process` parameter name shown here:

```
Get-Process | ForEach-Object -Process {
    Write-Host $_.Name -ForegroundColor Green
}
```

The script block executes once for each object in the input pipeline, that is, once for each object returned by `Get-Process`. The special variable `$_` is used to access the current object.

The variable `$PSItem` may be used in place of `$_` if desired. There is no difference between the two variable names. `$_` is the more commonly used name.

The `Process` parameter is accompanied by the `Begin` and `End` parameters. `Begin` and `End` are used to run script blocks before the first value is sent to the `Process` script block, and after the last value has been received.

Begin and End parameters

If `ForEach-Object` is given a single script block as an argument, it is passed to the `Process` parameter. The `Process` script block runs once for each object in the input pipeline.

`ForEach-Object` also supports `Begin` and `End` parameters. `Begin` runs once before the first value in the pipeline is passed. `End` runs once after the last value in the input pipeline.

The behavior of these parameters is shown in the following example:

```
1..5 | ForEach-Object -Begin {
    Write-Host "Starting the pipeline. Creating value."
    $value = 0
} -Process {
    Write-Host "Adding $_ to value."
    $value += $_
} -End {
    Write-Host "Finished the pipeline. Displaying value."
    $value
}
```

Positional parameters

The `ForEach-Object` command is written to allow all parameters to be passed based on position. The first positional parameter is `Process`. However, `ForEach-Object` will switch parameters around based on the number of arguments:

```
1 | ForEach-Object { "Process: $_" }
```

If more than one script block is passed, the first position is passed to the `Begin` parameter:

```
1 | ForEach-Object { "Begin" } { "Process: $_" }
```

If a third script block is added, it will be passed to the `End` parameter:

```
1 | ForEach-Object { "Begin" } { "Process: $_" } { "End" }
```

The Parallel parameter

In PowerShell 7, `ForEach-Object` gains a `Parallel` parameter. This, as the name suggests, can be used to run process blocks in parallel rather than one after another.

By default, `ForEach-Object` runs 5 instances of the process block at a time; this is controlled by the `ThrottleLimit` parameter. The limit may be increased (or decreased) depending on where the bottleneck is with a given process.

Running a simple `ForEach-Object` command with a `Start-Sleep` statement shows how the output is grouped together as each set of jobs completes:

```
1..10 | ForEach-Object -Parallel {
    Start-Sleep -Seconds 2
    $_
}
```

When using `ForEach-Object` without the `Parallel` parameter, variables created before the command are accessible without any special consideration:

```
$string = 'Hello world'
1 | ForEach-Object {
    # The string variable can be used as normal
    $string
}
```

When using `Parallel`, each parallel script block runs in a separate thread. Variables created outside the `ForEach-Object` command must be accessed by prefixing the variable name with the `using:` scope modifier as shown here:

```
$string = 'Hello world'
1 | ForEach-Object -Parallel {
    # The $string variable is only accessible if using is used.
    $using:string
}
```

For the most part, the `$using` scope modifier is one-way. That is, values may be read from the scope, but new values cannot be set. For example, the following attempt to write a value to `$using:newString` will fail:

```
PS> 1 | ForEach-Object -Parallel {
>>     $using:newString = $_
>> }
ParserError:
Line |
   2 |      $using:newString = $_
     |      ~~~~~~~~~~~~~~~~
     | The assignment expression is not valid. The input to an assignment
     | operator must be an object that is able to accept assignments,
     | such as a variable or a property.
```

Advanced array types and hashtables can be changed; however, parentheses are required around the variable name. For example:

```
$values = @{}
1..5 | ForEach-Object -Parallel {
    ($using:values).Add($_, $_)
}
$values
```

While it is possible to store values like this, the hashtable used previously is not built to be changed from multiple threads, as it is in the example. Hashtables will be discussed in *Chapter 5, Variables, Arrays, and Hashtables*. Generally speaking, output from a parallel `ForEach-Object` should be sent to the output pipeline; for example:

```
$output = 1..5 | ForEach-Object -Parallel { $_ }
```

`ForEach-Object` is most often used to add complexity when processing the output from another command, or when working on a collection. `ForEach-Object` may also be used to read a single property, or execute a method of every object in a collection.

The MemberName parameter

`ForEach-Object` may also be used to get a single property or execute a single method on each of the objects. For example, `ForEach-Object` may be used to return only the `Path` property when using `Get-Process`:

```
Get-Process | ForEach-Object -MemberName Path
```

Or, `ForEach-Object` may be used to run the `ToString` method on a set of dates:

```
PS> @(
>>     Get-Date '01/01/2019'
>>     Get-Date '01/01/2020'
>> ) | ForEach-Object ToString('yyyyMMdd')
20190101
20200101
```

`ForEach-Object` offers a great deal of flexibility.

The Where-Object command

Filtering the output from commands may be performed using `Where-Object`. For example, processes might be filtered for those started after 9am today. If there are no processes started after 9am, then the statement will not return anything:

```
Get-Process | Where-Object StartTime -gt (Get-Date 9:00:00)
```

The syntax shown in help for `Where-Object` does not quite match the syntax used here. The help text is as follows:

```
Where-Object [-Property] <String> [[-Value] <Object>] -GT ...
```

In the preceding example, we see the following:

- `StartTime` is the argument for the `Property` parameter (first argument by position)
- The comparison is greater than, as signified by the `gt` switch parameter
- The date (using the `Get-Date` command) is the argument for the `Value` parameter (second argument by position)

Based on that, the example might be written as follows:

```
Get-Process |
   Where-Object -Property StartTime -Value (Get-Date 9:00:00) -gt
```

However, it is far easier to read `StartTime` is greater than `<some date>`, so most examples tend to follow that pattern.

`Where-Object` also accepts filters using the `FilterScript` parameter. `FilterScript` is often used to describe more complex filters, filters where more than one term is used:

```
Get-Service | Where-Object {
    $_.StartType -eq 'Manual' -and
    $_.Status -eq 'Running'
}
```

When a filter like this is used, the conditions are evaluated in the order they are written in. This can be used to avoid conditions that may otherwise cause errors.

In the following example, `Test-Path` is used before `Get-Item`, which is used to test the last time a file was written on a remote computer (via the administrative share):

```
$date = (Get-Date).AddDays(-90)
'Computer1', 'Computer2' | Where-Object {
    (Test-Path "\\$_\c$\temp\file.txt") -and
    (Get-Item "\\$_\c$\temp\file.txt").LastWriteTime -lt $date
}
```

If `Test-Path` is removed, the snippet will throw an error if either the computer or the file does not exist.

Selecting and sorting

`Select-Object` acts on an input pipeline; either an existing collection of objects, or the output from another command. `Select-Object` can be used to select a subset of properties, to change properties, or to add new properties. `Select-Object` also allows a user to limit the number of objects returned.

`Sort-Object` can be used to perform both simple and complex sorting based on an input pipeline.

Using `Select-Object` is a key part of working with PowerShell, output can be customized to suit circumstances in a number of ways.

The Select-Object command

`Select-Object` is an extremely versatile command as shown by each of the following short examples, which demonstrate some of the simpler uses of the command.

`Select-Object` may be used to limit the properties returned by a command by name:

```
Get-Process | Select-Object -Property Name, Id
```

`Select-Object` can limit the properties returned from a command using wildcards:

```
Get-Process | Select-Object -Property Name, *Memory
```

`Select-Object` can list everything but a few properties:

```
Get-Process | Select-Object -Property * -ExcludeProperty *Memory*
```

`Select-Object` can get the first few objects in a pipeline:

```
Get-ChildItem C:\ -Recurse | Select-Object -First 2
```

Or `Select-Object` can select the last few objects in a pipeline:

```
Get-ChildItem C:\ | Select-Object -Last 3
```

`Select-Object` can `Skip` items at the beginning – in this example, the fifth item:

```
Get-ChildItem C:\ | Select-Object -Skip 4 -First 1
```

Or it can `Skip` items at the end. This example returns the third from the end:

```
Get-ChildItem C:\ | Select-Object -Skip 2 -Last 1
```

`Select-Object` can get an object from a pipeline by index:

```
Get-ChildItem C:\ | Select-Object -Index 3, 4, 5
```

In PowerShell 7 and above, `Select-Object` can omit certain indexes:

```
Get-ChildItem C:\ | Select-Object -SkipIndex 3, 4, 5
```

`Select-Object` also offers a number of more advanced uses. The following sections describe calculated properties, the `ExpandProperty` parameter, the `Unique` parameter, and property sets.

Calculated properties are perhaps one of the most commonly used features of `Select-Object`.

Calculated properties

`Select-Object` can be used to add new properties to or rename existing properties on objects in an input pipeline.

Calculated properties are described using a hashtable with specific key names. The format is described in help for `Select-Object`. In addition to names, the following hashtable formats are acceptable for the `Property` parameter:

```
@{ Name = 'PropertyName'; Expression = { 'PropertyValue' } }
@{ Label = 'PropertyName'; Expression = { 'PropertyValue' } }
@{ n = 'PropertyName'; e = { 'PropertyValue' } }
@{ l = 'PropertyName'; e = { 'PropertyValue' } }
```

The expression is most often a script block, this allows other commands to be executed, mathematical operations to be performed, substitutions to be made, and so on.

If a property is being renamed, a quoted string can be used instead of the script block. The following two examples have the same result:

```
Get-Process | Select-Object @{ Name = 'ProcessID'; Expression = 'ID' }
Get-Process | Select-Object @{ Name = 'ProcessID'; Expression = { $_.ID } }
```

If the list of properties is long, it can be better to enclose the list in `@()`, the array operator, allowing the properties to be spread across different lines:

```
Get-Process | Select-Object -Property @(
    'Name'
    @{Name = 'ProcessId'; Expression = 'ID' }
    @{Name = 'FileOwner'; Expression = { (Get-Acl $_.Path).Owner }}
)
```

Any number of properties might be added in this manner. The preceding example includes the output from `Get-Acl`; if more than one property were required, `ForEach-Object` might be added to the command:

```
Get-Process | Where-Object Path | ForEach-Object {
    $acl = Get-Acl $_.Path
    Select-Object -InputObject $_ -Property @(
        'Name'
        @{Name = 'ProcessId'; Expression = 'ID' }
        @{Name = 'FileOwner'; Expression = { $acl.Owner }}
        @{Name = 'Access';    Expression = { $acl.AccessToString }}
    )
}
```

When `Select-Object` is used with the `Property` parameter, a new custom object is always created. If the existing object type must be preserved, `Add-Member` should be used instead. For example, if the object includes methods that must be accessible later in a script, the `Property` parameter of `Select-Object` should not be used.

The following example shows that the object type is preserved if the `Property` parameter is not used. The following command shows that the type is `Process`, which allows any of the methods, such as `WaitForExit`, to be used later on in a script:

```
(Get-Process | Select-Object -First 1).GetType()
```

If the `Property` parameter is used, the output is the `PSCustomObject` type. The resulting object will not have any of the methods specific to the `Process` type:

```
(Get-Process | Select-Object -Property Path, Company -First 1).GetType()
```

Calculated properties are extremely flexible, allowing an object to be modified, or a more complex object to be created with a relatively small amount of code.

The ExpandProperty parameter

The `ExpandProperty` parameter of `Select-Object` may be used to expand a single property of an object. This might be used to expand a property containing a string, leaving the output as an array of strings:

```
Get-Process | Select-Object -First 5 -ExpandProperty Name
```

If `ExpandProperty` is omitted, the returned object will be a `PSCustomObject` object with a `Name` property rather than the simpler array of strings.

Expanding a single property containing a string, or a numeric value, or an array of either, is the most common use of the `ExpandProperty` parameter.

Occasionally, it may be desirable or necessary to expand a property containing a more complex object. The members of the expanded property are added to the custom object:

```
Get-ChildItem $env:SYSTEMROOT\*.dll |
    Select-Object -Property FullName, Length -ExpandProperty VersionInfo |
    Format-List *
```

Conflicting member names will cause an error to be raised; the conflicting name is otherwise ignored.

It is possible, if unusual, to use this technique to build up a single custom object based on the output from multiple commands:

```
Get-CimInstance Win32_ComputerSystem |
    Select-Object -Property @(
        @{n='ComputerName';e={ $_.Name }}
        'DnsHostName'
        @{n='OSInfo';e={ Get-CimInstance Win32_OperatingSystem }}
    ) |
    Select-Object * -ExpandProperty OSInfo |
    Select-Object -Property @(
        'ComputerName'
        'DnsHostName'
        @{n='OperatingSystem';e='Caption'}
        'SystemDirectory'
    )
```

The resulting object will contain two properties from `Win32_ComputerSystem` and two properties from `Win32_OperatingSystem`.

The Unique parameter

`Select-Object` returns unique values from arrays of simple values with the `Unique` parameter:

```
1, 1, 1, 3, 5, 2, 2, 4 | Select-Object -Unique
```

About Get-Unique

`Get-Unique` may also be used to create a list of unique elements. When using `Get-Unique`, a list must be sorted first; for example:

```
1, 1, 1, 3, 5, 2, 2, 4 | Sort-Object | Get-Unique
```

In the following example, an object is created with one property called `Number`. The value for the property is `1`, `2`, or `3`. The result is two objects with a value of `1`, two with a value of `2`, and so on:

```
PS> 1, 2, 3, 1, 2, 3 | ForEach-Object {
>>     [PSCustomObject]@{
>>         Number = $_
>>     }
>> }
Number
------
     1
     2
     3
     1
     2
     3
```

`Select-Object` can remove the duplicates from the set in this example using the `-Unique` parameter if a list of properties (or a wildcard for the properties) is set:

```
PS> 1, 2, 3, 1, 2, 3 |
>>    ForEach-Object {
>>        [PSCustomObject]@{
>>            Number = $_
>>        }
>>    } |
>>    Select-Object -Property * -Unique
Number
------
     1
     2
     3
```

`Select-Object` builds up a collection of unique objects by comparing each property of each object to every unique object that came before it in a pipeline. This allows `Select-Object` to work without relying on an ordered collection (as `Get-Unique` requires) but is consequently slower; the work of comparing increases every time a unique object is encountered.

Property sets

A property set is a pre-defined list of properties that might be used when exploring an object. The property set is stored with the object itself. `Select-Object` can be used to select the properties within a specified property set.

In the following example, `Get-Member` is used to view property sets available on the objects returned by `Get-Process`:

```
PS> Get-Process -Id $PID | Get-Member -MemberType PropertySet
    TypeName: System.Diagnostics.Process
Name               MemberType     Definition
----               ----------     ----------
PSConfiguration    PropertySet    PSConfiguration {Name, Id, ...
PSResources        PropertySet    PSResources {Name, Id, Hand...
```

`Select-Object` may then be used to display one of the property sets, `PSConfiguration`:

```
PS> Get-Process -Id $PID | Select-Object -Property PSConfiguration
Name   Id PriorityClass FileVersion
----   -- ------------- -----------
pwsh 2220        Normal 7.0.0.0
```

The Sort-Object command

The `Sort-Object` command allows objects to be sorted. By default, `Sort-Object` will sort objects in ascending order:

```
PS> 5, 4, 3, 2, 1 | Sort-Object
1
2
3
4
5
```

Strings are sorted in ascending order, irrespective of uppercase or lowercase:

```
PS> 'ccc', 'BBB', 'aaa' | Sort-Object
aaa
BBB
ccc
```

When dealing with more complex objects, `Sort-Object` may be used to sort based on a named property. For example, processes may be sorted based on the `Id` property:

```
Get-Process | Sort-Object -Property Id
```

Objects may be sorted on multiple properties; for example, a list of files may be sorted on `LastWriteTime` and then on `Name` :

```
Get-ChildItem C:\Windows\System32 |
    Sort-Object LastWriteTime, Name
```

In the preceding example, items are first sorted on `LastWriteTime` . Items that have the same value for `LastWriteTime` are then sorted based on `Name` .

`Sort-Object` is not limited to sorting on existing properties. A script block (a fragment of script, enclosed in curly braces) can be used to create a calculated value for sorting. For example, it is possible to order items based on a word, as shown in this example:

```
$examResults = @(
    [PSCustomObject]@{ Exam = 'Music';   Result = 'N/A';  Mark = 0 }
    [PSCustomObject]@{ Exam = 'History'; Result = 'Fail'; Mark = 23 }
    [PSCustomObject]@{ Exam = 'Biology'; Result = 'Pass'; Mark = 78 }
    [PSCustomObject]@{ Exam = 'Physics'; Result = 'Pass'; Mark = 86 }
    [PSCustomObject]@{ Exam = 'Maths';   Result = 'Pass'; Mark = 92 }
)
$examResults | Sort-Object {
    switch ($_.Result) {
        'Pass' { 1 }
        'Fail' { 2 }
        'N/A'  { 3 }
    }
}
```

The result of sorting the objects is shown here:

```
Exam      Result   Mark
----      ------   ----
Maths     Pass       92
Physics   Pass       86
Biology   Pass       78
History   Fail       23
Music      N/A        0
```

In the preceding example, when `Sort-Object` encounters a `Pass` result it is given the lowest numeric value (`1`) to sort on. As `Sort-Object` defaults to ascending order, this means exams with a result of `Pass` appear first in the list. This process is repeated to give a numeric value to each of the other possible results.

Sorting within each `Result` set varies depending on the version of PowerShell. Windows PowerShell changes the order of the elements within each set, listing `Maths`, `Physics`, then `Biology`. PowerShell 6 and above, on the other hand, maintains the original order, listing `Biology`, then `Physics`, then `Maths` within the pass set.

As `Sort-Object` is capable of sorting on more than one property, the preceding example can be taken further to sort on mark next. This makes the output order entirely predictable, regardless of the version of PowerShell:

```
PS> $examResults | Sort-Object {
>>     switch ($_.Result) {
>>         'Pass' { 1 }
>>         'Fail' { 2 }
>>         'N/A'  { 3 }
>>     }
>> }, Mark
Exam     Result    Mark
----     ------    ----
Biology    Pass      78
Physics    Pass      86
Maths      Pass      92
History    Fail      23
Music       N/A       0
```

Adding the `Descending` parameter to `Sort-Object` will reverse the order of both fields:

```
PS> $examResults | Sort-Object {
>>     switch ($_.Result) {
>>         'Pass' { 1 }
>>         'Fail' { 2 }
>>         'N/A'  { 3 }
>>     }
>> }, Mark -Descending
Exam     Result    Mark
----     ------    ----
Music       N/A       0
History    Fail      23
Maths      Pass      92
Physics    Pass      86
Biology    Pass      78
```

The ordering behavior can be made property-specific using the notation that is shown in the following example:

```
PS> $examResults | Sort-Object {
>>     switch ($_.Result) {
>>         'Pass' { 1 }
>>         'Fail' { 2 }
>>         'N/A'  { 3 }
>>     }
>> }, @{ Expression = { $_.Mark }; Descending = $true }
Exam     Result    Mark
----     ------    ----
Maths      Pass      92
Physics    Pass      86
Biology    Pass      78
History    Fail      23
Music       N/A       0
```

The hashtable, `@{}`, is used to describe an expression (a calculated property; in this case, the value for `Mark`) and the sorting order, which is either ascending or descending.

In the preceding example, the first sorting property, based on the `Result` property, is sorted in ascending order as this is the default. The second property, `Mark`, is sorted in descending order.

Once a set of data is has been prepared by selecting and sorting, grouping and measuring can be used to work on the collection.

Grouping and measuring

`Group-Object` is a powerful command that allows you to group objects together based on a property or expression.

`Measure-Object` supports several simple mathematical operations, such as counting the number of objects, calculating an average, calculating a sum, and so on. `Measure-Object` also allows characters, words, or lines to be counted in text fields.

The Group-Object command

The `Group-Object` command shows a group and count for each occurrence of a value in a collection of objects.

Given the sequence of numbers shown, `Group-Object` creates a `Name` that holds the value it is grouping, a `Count` as the number of occurrences of that value, and a `Group` property as the set of similar values:

```
PS> 6, 7, 7, 8, 8, 8 | Group-Object
Count    Name                      Group
-----    ----                      -----
    1    6                         {6}
    2    7                         {7, 7}
    3    8                         {8, 8, 8}
```

The `Group` property may be removed using the `NoElement` parameter, which simplifies the output of the command:

```
PS> 6, 7, 7, 8, 8, 8 | Group-Object -NoElement
Count    Name
-----    ----
    1    6
    2    7
    3    8
```

`Group-Object` can group based on a specific property. For example, it might be desirable to list the number of occurrences of a file in an extensive folder structure. In the following example, the `C:\Windows\Assembly` folder contains different versions of DLLs for different versions of packages, including the .NET Framework:

```
Get-ChildItem C:\Windows\Assembly -Filter *.dll -Recurse |
    Group-Object Name
```

Combining `Group-Object` with commands such as `Where-Object` and `Sort-Object` allows reports about the content of a set of data to be generated extremely quickly, for example, a report on the names of the top five files that appear more than once in a file tree:

```
Get-ChildItem C:\Windows\Assembly -Filter *.dll -File -Recurse |
    Group-Object Name -NoElement |
    Where-Object Count -gt 1 |
    Sort-Object Count, Name -Descending |
    Select-Object Name, Count -First 5
```

The output from the preceding command will vary from one computer to another; it depends on the installed software, development kits, .NET Framework version, and so on. The output from the preceding command might be similar to the following example:

```
Name                                            Count
----                                            -----
Microsoft.Web.Diagnostics.resources.dll         14
Microsoft.Web.Deployment.resources.dll          14
Microsoft.Web.Deployment.PowerShell.resources.dll    14
Microsoft.Web.Delegation.resources.dll          14
Microsoft.Web.PlatformInstaller.resources.dll   13
```

As was seen with `Sort-Object`, `Group-Object` can group on more than one property. For example, we might group on both a filename and the size of a file (the `Length` property of a file):

```
Get-ChildItem C:\Windows\Assembly -Filter *.dll -Recurse |
    Group-Object Name, Length -NoElement |
    Where-Object Count -gt 1 |
    Sort-Object Name -Descending |
    Select-Object Name, Count -First 5
```

As with the previous example, the output from the command will vary from one computer to another:

```
Name                                                   Count
----                                                   -----
WindowsFormsIntegration.Package.ni.dll, 100352             2
Templates.Editorconfig.Wizard.resources.ni.dll, 9216      13
Templates.Editorconfig.Wizard.resources.ni.dll, 8192      13
System.Web.ni.dll, 16939008                                2
System.Web.ni.dll, 14463488                                2
```

In the preceding example, `System.Web.ni.dll` is listed twice (with a count of two in each case). Each pair of files has the same file size.

Like `Sort-Object`, `Group-Object` is not limited to properties that already exist. Calculated properties can be used to create a new value to group. For example, grouping on an email domain in a list of email addresses might be useful. The domain is obtained by splitting on the `@` character:

```
PS> 'one@one.example', 'two@one.example', 'three@two.example' |
>>     Group-Object { ($_ -split '@')[1] }
Count    Name                      Group
-----    ----                      -----
    2    one.example               {one@one.example, two@one.example}
    1    two.example               {three@two.example}
```

In this example, the split operator is used to split on the `@` character; everything to the left is stored in index `0`, while everything to the right is stored in index `1`.

By default, `Group-Object` returns the collection of objects shown in each of the preceding examples. `Group-Object` can also return a hashtable using the `AsHashtable` parameter.

When using the `AsHashTable` parameter, you can add the `AsString` parameter. For more complex values, this ensures the groups can be accessed. The `AsString` parameter forces the key for each entry in the hashtable to be a string; for example:

```
$hashtable = @(
    [IPAddress]'10.0.0.1'
    [IPAddress]'10.0.0.2'
    [IPAddress]'10.0.0.1'
) | Group-Object -AsHashtable -AsString
$hashtable['10.0.0.1']
```

If the `AsString` parameter was excluded from the preceding example, the value used to access the key would have to be an `IPAddress` type. For example:

```
$hashtable = @(
    [IPAddress]'10.0.0.1'
    [IPAddress]'10.0.0.2'
    [IPAddress]'10.0.0.1'
) | Group-Object -AsHashtable
$hashtable[[IPAddress]'10.0.0.1']
```

In the preceding example, attempting to access the key without the `[IPAddress]` type will fail; no value will be returned, and no error will be shown.

By default, `Group-Object` is not case sensitive. The strings `one`, `ONE`, and `One` are all considered equal when grouping. The `CaseSensitive` parameter forces `Group-Object` to differentiate between items where cases differ:

```
PS> 'one', 'ONE', 'One' | Group-Object -CaseSensitive
Count Name                      Group
----- ----                      -----
    1 one                       {one}
    1 ONE                       {ONE}
    1 One                       {One}
```

`Group-Object` can be used to count occurrences of a value within a collection of objects. `Measure-Object` is useful when it is necessary to analyze the values, such as when determining the average of a specific property.

The Measure-Object command

When used without any parameters, `Measure-Object` returns a value for `Count`, which is the number of items passed in using the pipeline; for example:

```
PS> 1, 5, 9, 79 | Measure-Object
Count     : 4
Average   :
Sum       :
Maximum   :
Minimum   :
Property  :
```

Each of the remaining properties is empty, unless requested using their respective parameters. For example, `Sum` may be requested:

```
PS> 1, 5, 9, 79 | Measure-Object -Sum
Count      : 4
Average    :
Sum        : 94
Maximum    :
Minimum    :
Property   :
```

Adding the remaining parameters adds values to the rest of the fields (except `Property`):

```
PS> 1, 5, 9, 79 | Measure-Object -Average -Maximum -Minimum -Sum
Count       : 4
Average     : 23.5
Sum         : 94
Maximum     : 79
Minimum     : 1
Property    :
```

The value for `Property` is added when `Measure-Object` is asked to work against a particular property (instead of a set of numbers). For example:

```
PS> Get-Process | Measure-Object WorkingSet -Average
Count        : 135
Average      : 39449395.2
Sum          :
Maximum      :
Minimum      :
Property     : WorkingSet
```

When working with text, `Measure-Object` can count characters, words, or lines. For example, it can be used to count the number of lines, words, and characters in a text file:

```
PS> Get-Content C:\Windows\WindowsUpdate.log |
>>    Measure-Object -Line -Word -Character
Lines    Words    Characters    Property
-----    -----    ----------    --------
    3       32           268
```

`Group-Object` and `Measure-Object` are essential parts of a PowerShell toolkit. They can significantly simplify analyzing collections of data for repetition or performing simple mathematical operations. Each of these commands is used against a single collection of objects; when working with more than one collection of objects, it may be necessary to compare.

Comparing

You can use the `Compare-Object` command to compare collections of objects with one another.

`Compare-Object` must be supplied with values for the `ReferenceObject` and `DifferenceObject` parameters, which are normally collections or arrays of objects. If either value is null, then an error will be displayed. If both values are equal, `Compare-Object` does not return anything by default. For example, the reference and difference objects in the following example are identical:

```
Compare-Object -ReferenceObject 1, 2 -DifferenceObject 1, 2
```

If there are differences, `Compare-Object` displays the results, as shown here:

```
PS> Compare-Object -ReferenceObject 1, 2, 3, 4 -DifferenceObject 1, 2
InputObject SideIndicator
----------- -------------
          3 <=
          4 <=
```

This shows that `ReferenceObject` (the collection on the left, denoted by the direction of the `<=` arrow) has the values, but `DifferenceObject` (the collection on the right) does not.

`Compare-Object` has several other parameters that may be used to change the output. The `IncludeEqual` parameter adds values that are present in both collections to the output:

```
PS> $params = @{
>>     ReferenceObject  = 1, 2, 3, 4
>>     DifferenceObject = 1, 2
>>     IncludeEqual     = $true
>> }
PS> Compare-Object @params

InputObject SideIndicator
----------- -------------
          1 ==
          2 ==
          3 <=
          4 <=
```

`ExcludeDifferent` will omit the results that differ. This parameter makes sense if `IncludeEqual` is also set; without this, the command will always return nothing.

The `PassThru` parameter is used to return the original object instead of the representation showing the differences. In the following example, it is used to select values that are common to both the reference and difference objects:

```
PS> $params = @{
>>     ReferenceObject  = 1, 2, 3, 4
>>     DifferenceObject = 1, 2
>>     ExcludeDifferent = $true
>>     IncludeEqual     = $true
>>     PassThru         = $true
>> }
PS> Compare-Object @params
1
2
```

`Compare-Object` can compare based on properties of objects, as well as the simpler values in the preceding examples. This can be a single property, or a list of properties. For example, the following command compares the contents of `C:\Windows\System32` with `C:\Windows\SysWOW64`, returning files that have the same name and are the same size in both:

```
$params = @{
    ReferenceObject  = Get-ChildItem C:\Windows\System32 -File
    DifferenceObject = Get-ChildItem C:\Windows\SysWOW64 -File
    Property         = 'Name', 'Length'
    IncludeEqual     = $true
    ExcludeDifferent = $true
}
Compare-Object @params
```

By default, `Compare-Object` writes an error if either the reference or difference objects are null. If `Compare-Object` is used when there is a chance of either being empty, the following technique can be used to avoid an error being generated provided neither contains an explicit null value:

```
$reference = Get-ChildItem C:\Windows\System32\tcpmon*.ini
$difference = Get-ChildItem C:\Windows\SysWOW64\tcpmon*.ini
Compare-Object @($reference) @($difference) -Property Name
```

The array operator (`@()`) around each parameter value will be discarded by PowerShell. If `$difference` is empty, it will be treated as an empty array instead of it being a null value.

Collections of objects generated by any of the preceding commands might be exported to move data outside of PowerShell. The result of any of these operations might be exported to a file for another user, system, or program to use.

Importing, exporting, and converting

Getting data in and out of PowerShell is a critical part of using the language. There are several commands dedicated to this task, including:

- `Export-Csv`
- `Import-Csv`
- `Export-CliXml`
- `Import-CliXml`
- `Tee-Object`

Other PowerShell modules, such as the `ImportExcel` module, available in the PowerShell Gallery can be used to extend the output formats available. Commands such as `ConvertTo-Html`, `ConvertTo-Json`, and `ConvertFrom-Json` are explored in later chapters.

The Export-Csv command

The `Export-Csv` command writes data from objects to a text file; for example:

```
Get-Process | Export-Csv processes.csv
```

By default, `Export-Csv` writes a comma-delimited file using UTF8 encoding and completely overwrites any file using the same name.

`Export-Csv` may be used to add lines to an existing file using the `Append` parameter. When the `Append` parameter is used, the input object must have each of the fields listed in the CSV header or an error will be thrown unless the `Force` parameter is used:

```
PS> Get-Process -ID $PID |
>>     Select-Object Name, Id |
>>     Export-Csv .\Processes.csv
PS> Get-Process explorer |
>>     Select-Object Name |
>>     Export-Csv .\Processes.csv -Append
Export-Csv: Cannot append CSV content to the following file: .\Processes.csv. The appended object does
not have a property that corresponds to the following column: Id. To continue with mismatched
properties, add the -Force parameter, and then retry the command.
```

If you use the `Append` parameter and the input object has more fields than the CSV, the extra fields are silently dropped when writing the CSV file. For example, the value held in `Id` is ignored when writing the results to the existing CSV file:

```
Get-Process pwsh |
    Select-Object Name | Export-Csv .\Processes.csv
Get-Process explorer |
    Select-Object Name, Id |
    Export-Csv .\Processes.csv -Append
```

`Export-Csv` in Windows PowerShell writes a header line to each file, which details the .NET type it has just exported. If the preceding example was used in Windows PowerShell, the header line would be as follows:

```
#TYPE Selected.System.Diagnostics.Process
```

This header is only included in PowerShell 7 (and PowerShell Core) when explicitly using the `IncludeTypeInformation` parameter.

`Export-Csv` in Windows PowerShell can be instructed to exclude this header using the `NoTypeInformation` parameter:

```
Get-Process | Export-Csv processes.csv -NoTypeInformation
```

`ConvertTo-Csv` in Windows PowerShell is like `Export-Csv`, except that instead of writing content to a file, content is written as command output:

```
PS> Get-Process powershell | Select-Object Name, Id | ConvertTo-Csv
#TYPE Selected.System.Diagnostics.Process
"Name","Id"
"pwsh","404"
```

Both `Export-Csv` and `ConvertTo-Csv` are limited in what they can do with arrays of objects in properties. For example, `ConvertTo-Csv` is unable to display the values that are in an array:

```
PS> [PSCustomObject]@{
>>     Name = "Numbers"
>>     Value = 1, 2, 3, 4, 5
>> } | ConvertTo-Csv -NoTypeInformation
"Name","Value"
"Numbers","System.Object[]"
```

The value of the `Value` property in the CSV content is taken from the `ToString` method, which is called on the property named `Value`; for example:

```
PS> $object = [PSCustomObject]@{
>>     Name = "Numbers"
>>     Value = 1, 2, 3, 4, 5
>> }
PS> $object.Value.ToString()
System.Object[]
```

If a CSV file is expected to hold the content of an array, code must be written to convert it into a suitable format. For example, the content of the array can be written after converting it to a string:

```
PS> [PSCustomObject]@{
>>     Name  = "Numbers"
>>     Value = 1, 2, 3, 4, 5
>> } | ForEach-Object {
>>     $_.Value = $_.Value -join ', '
>>     $_
>> } | ConvertTo-Csv -NoTypeInformation
"Name","Value"
"Numbers","1, 2, 3, 4, 5"
```

In the preceding example, the value of the property is joined using a comma followed by a space. The modified object (held in `$_`) is passed on to the `ConvertTo-Csv` command.

The Import-Csv command

Comma-Separated Value (**CSV**) files are structured text. Applications such as Microsoft Excel can work with CSV files without changing the file format, although Excel's advanced features cannot be saved to a CSV file.

By default, `Import-Csv` expects the input to have a header row, be comma-delimited, and use ASCII file encoding. If any of these items are different, the `command` parameters may be used. For example, a tab may be set as the delimiter:

```
Import-Csv TabDelimitedFile.tsv -Delimiter `t
```

A tick (grave accent) followed by *t* (`` `t ``) is used to represent the tab character in PowerShell.

Data imported using `Import-Csv` will always be formatted as a string. If `Import-Csv` is used to read a file containing the following text, each of the numbers will be treated as a string:

```
Name,Position
Jim,35
Matt,3
Dave,5
```

Attempting to use `Sort-Object` on the imported CSV file results in values being sorted as if they were strings, not numbers:

```
PS> Import-Csv .\positions.csv | Sort-Object Position
Name    Position
----    --------
Matt    3
Jim     35
Dave    5
```

You can use `Sort-Object` to consider the value for `Position` as an integer by using a script block expression:

```
PS> Import-Csv .\positions.csv | Sort-Object { [Int]$_.Position }
Name Position
---- --------
Matt  3
Dave  5
Jim   35
```

This conversion problem exists regardless of whether the data in a CSV file is numeric, a date, or any type other than a string.

The `ConvertFrom-Csv` command is similar to `Import-Csv` in that it reads CSV content and creates custom objects from that content. The difference is that `ConvertFrom-Csv` reads strings from standard input instead of a file. In the following example, a string is converted into a custom object using the `ConvertFrom-Csv` command with the `Header` parameter:

```
PS> "powershell,404" | ConvertFrom-Csv -Header Name, Id
Name         Id
----         --
powershell   404
```

`ConvertFrom-Csv` expects either an array of strings or a single string with line-breaks. The following example includes a header in the string:

```
"Name,Id
Powershell,404" | ConvertFrom-Csv
```

CSV is a simple and accessible format. However, CSV is a pure-text format; it cannot express different value types (such as numbers and dates) – all data is a string. The CliXml format is at the other end of the spectrum: it can be used to store complex data types.

Export-Clixml and Import-Clixml

`Export-Clixml` creates representations of objects in XML files. The CLI acronym stands for Common Language Infrastructure, a technical standard developed by Microsoft. `Export-Clixml` is extremely useful where type information about each property must be preserved.

For example, the following object may be exported using `Export-Clixml`:

```
[PSCustomObject]@{
    Integer = 1
    Decimal = 2.3
    String  = 'Hello world'
} | Export-Clixml .\object.xml
```

The resulting XML file shows the type for each of the properties it has just exported:

```
PS> Get-Content object.xml
<Objs Version="1.1.0.1" xmlns="http://schemas.microsoft.com/powershell/2004/04">
  <Obj RefId="0">
    <TN RefId="0">
      <T>System.Management.Automation.PSCustomObject</T>
      <T>System.Object</T>
    </TN>
    <MS>
      <I32 N="Number">1</I32>
      <Db N="Decimal">2.3</Db>
      <S N="String">Hello world</S>
    </MS>
  </Obj>
</Objs>
```

In the preceding example, `I32` is a 32-bit integer (Int32). `Db` is a double-precision floating-point number (double). `S` is a string.

With this extra information in the file, PowerShell can rebuild the object, including the different types, using `Import-Clixml`, as follows:

```
$object = Import-Clixml .\object.xml
```

Once imported, the value types can be inspected using the `GetType` method:

```
PS> $object.Decimal.GetType()
IsPublic    IsSerial    Name            BaseType
--------    --------    ----            --------
   True         True    Double          System.ValueType
```

The ability to rebuild the original object allows `Export-CliXml` to convert credential objects to text. When it does so, password values are encrypted using **Data Protection API** (**DPAPI**) on Windows. For example, providing a username and password when prompted will create an XML file holding the encrypted password:

```
Get-Credential | Export-CliXml -Path secret.xml
```

The file can be opened in a text editor to view the encrypted password. The credential can be imported again using `Import-CliXml`:

```
$credential = Import-CliXml -Path secret.xml
```

The password for the credential is encrypted using the current user account (protected by the login password). The key used is held in the user profile; the resulting file can only be decrypted on the computer it was created on (without a roaming profile).

The Tee-Object command

The `Tee-Object` command is used to send output to two places at the same time. `Tee-Object` is used to write output to a console and a file or variable.

For example, the following command both displays the output of `Get-Process` on the screen and writes the content to a

`$processes` variable.

```
Get-Process | Tee-Object -Variable processes
```

`Tee-Object` writes file output as the console sees it (table or list) rather than writing in CSV or another format.

Summary

The pipeline is a key component of PowerShell. It allows data, as objects, to be sent from one command to another. Each command can act on the data it has received and, in many cases, return more data.

PowerShell includes a variety of commands for working with objects in a pipeline. These commands are used again and again no matter how experienced a PowerShell developer might be.

The `Get-Member` command allows the members (properties, methods, and so on) to be explored, which can be used to understand what an object is capable of.

`ForEach-Object` is a common command used to run arbitrary code against objects in a pipeline. `Where-Object` may be used to filter a pipeline, returning only relevant objects.

The `Select-Object` command is used to define what properties should be returned. You can also use `Select-Object` to include or remove objects, for example, by selecting the first few objects from a pipeline. The `Sort-Object` command takes pipeline input and allows it to be sorted based on property names, or more complex criteria described by a script block.

You can perform comparisons of collections of objects using the `Compare-Object` command.

Finally, content in PowerShell can be exported to and imported from files in a variety of different ways. This chapter explored exporting and importing content using the `Export-Csv` and `Import-Csv` commands as well as the more complex output created by `Export-CliXml`.

The next chapter, *Chapter 4*, *Operators*, explores the wide variety of operators available in PowerShell, ranging from arithmetic and comparison operators to binary and redirection operators.

4

Operators

In programming, an operator is an object that is used to manipulate an item of data. Operations include comparing values, performing replacement and split operations, mathematical operations, bitwise operations, and so on. An operator is a fundamental part of any programming language and PowerShell is no exception.

PowerShell has a wide variety of operators; each one is explored within this chapter.

This chapter covers the following topics:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Regular expression-based operators
- Logical operators
- Binary operators
- Type operators
- Redirection operators
- Other operators

Arithmetic operators

Arithmetic operators are used to perform numeric calculations. The arithmetic operators in PowerShell are as follows:

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Remainder: `%`

As well as its use in numeric calculations, the addition operator may also be used with strings, arrays, and hashtables, and the multiplication operator may also be used with strings and arrays.

The following sections explore each of the operators listed previously.

Operator precedence

Operations are executed in a specific order depending on the operator used. For example, consider the following two simple calculations:

```
3 + 2 * 2
2 * 2 + 3
```

The result of both preceding expressions is 7 (2 multiplied by 2, then add 3) because the multiplication operator has higher precedence than the addition operator.

PowerShell includes a help document describing the precedence for each operator:

```
Get-Help about_Operator_Precedence
```

The help topic lists the operators in the order they are evaluated. The multiplication operator, `*`, is higher precedence (higher in the list) than addition, `+`; therefore, in an operation using both operators, `*` is calculated first.

Expressions in brackets have the highest precedence and are therefore always calculated before any other. Brackets may be used to group expressions together in cases where the default operator precedence would give an incorrect result:

```
(3 + 2) * 2
```

The result of the preceding calculation is 10; the expression in brackets is calculated first, giving 5, and the result of that is multiplied by 2.

Addition and subtraction operators

The addition and subtraction operators, `+` and `-`, are most easily recognizable as arithmetic operators. The addition operator also serves as a concatenation operator.

Addition operator

The addition operator may be used to add numeric values. For example, the following simple addition operation will result in the value `5.14159`:

```
2.71828 + 3.14159
```

The addition operator may also be used to concatenate strings:

```
'good' + 'bye'
```

If an attempt is made to concatenate a string with a number, the number will be converted into a string:

```
'hello number ' + 1
```

This style of operation fails if the number is used first. PowerShell expects the entire expression to be numeric if the left-hand side of the expression is numeric. PowerShell will try and convert the value on the right into a number and will fail if it can't convert the value to a number:

```
PS> 1 + ' is the number I like to use'
InvalidArgument: Cannot convert value "is the number I like to use" to type "System.Int32". Error:
"Input string was not in a correct format."
```

The addition operator may be used to add single elements to an existing array. As arrays are of fixed size, a new array will be created containing `1`, `2`, and `3`:

```
@(1, 2) + 3
```

Joining arrays with the addition operator is simple. Each of the following three examples creates an array and each array contains the values `1`, `2`, `3`, and `4`:

```
@(1, 2) + @(3, 4)
(1, 2) + (3, 4)
1, 2 + 3, 4
```

The last example is an array containing `1`, `2`, `3`, and `4` because the array operator has higher precedence than addition. Therefore, two arrays are added together.

Hashtables may be joined in a similar manner:

```
@{key1 = 1} + @{key2 = 2}
```

The addition operation fails if keys are duplicated as part of the addition operation:

```
PS> @{key1 = 1} + @{key1 = 2}
OperationStopped: Item has already been added. Key in dictionary: 'key1'  Key being added:
'key1'Subtraction operator
```

Subtraction operator

The subtraction operator may only be used for numeric expressions. The results of the following expressions are 3 and -18, respectively:

```
5 - 2
2 - 20
```

Subtraction is a simple but important operation used in everyday calculations. The following sections explore the multiplication, division, and remainder operators.

Multiplication, division, and remainder operators

Like the addition operator, the multiplication operator can act on strings. The division and remainder operators perform mathematical operations only.

Multiplication operator

The multiplication operator can perform simple numeric operations. For example, the result of the following expression is 5:

```
2.5 * 2
```

You can also use the multiplication operator to duplicate strings, such as in the following example that results in `hellohellohello`:

```
'hello' * 3
```

As with the addition operator, the multiplication operator will throw an error if a number is on the left of the expression. PowerShell will expect the entire expression to be numeric if the value on the left-hand side is numeric:

```
PS> 3 * 'hello'
InvalidArgument: Cannot convert value "hello" to type "System.Int32". Error: "Input string was not in
a correct format."
```

The multiplication operator may also be used to duplicate arrays. Each of the following examples creates an array containing `one`, `two`, `one`, and `two`:

```
@('one', 'two') * 2
('one', 'two') * 2
'one', 'two' * 2
```

Division operator

The division operator performs numeric division:

```
20 / 5
```

An error will be thrown if an attempt to divide by `0` is made:

```
PS> 1 / 0
RuntimeException: Attempted to divide by zero.
```

Division using negative numbers is permitted in PowerShell. When a positive number is divided by a negative number, the result is negative.

Remainder operator

The remainder operator returns the remainder of the whole-number (integer) division. For example, the result of the following operation is 1:

```
3 % 2
```

One possible use of the remainder operator is alternating, that is, swapping between two values repeatedly. This can be used to perform an action on every second, third, fourth, and x^{th} increment in an iteration. For example:

```
1..20 | ForEach-Object {
    if ($_ % 2 -eq 0) {
        $foregroundColor = 'Cyan'
    } else {
        $foregroundColor = 'White'
    }
    Write-Host $_ -ForegroundColor $foregroundColor
}
```

The preceding example alternates the color of the text.

Assignment operators

Assignment operators are used to give values to variables. The assignment operators that are available are as follows:

- Assign: `=`
- Add and assign: `+=`
- Subtract and assign: `-=`
- Multiply and assign: `*=`
- Divide and assign: `/=`
- Remainder and assign: `%=`

As with the arithmetic operators, add and assign may be used with strings, arrays, and hashtables. Multiply and assign may be used with strings and arrays.

Assign, add and assign, and subtract and assign

The assignment operator (`=`) is used to assign values to variables and properties; for example, it may be used to assign a value to a variable:

```
$variable = 'some value'
```

Or, the PowerShell console window title (or Windows Terminal tab title) might be changed by assigning a new value to the `WindowTitle` property:

```
$host.UI.RawUI.WindowTitle = 'PowerShell window'
```

The add and assign operator (`+=`) operates in a similar manner to the addition operator. The following example assigns the value `1` to a variable, then `+=` is used to add `20` to that value:

```
$i = 1
$i += 20
```

The preceding example is equivalent to writing the following:

```
$i = 1
$i = $i + 20
```

You can use the `+=` operator to concatenate strings:

```
$string = 'one'
$string += 'one'
```

As with the addition operator, attempting to add a numeric value to an existing string is acceptable. Attempting to add a string to a variable containing a numeric value is not:

```
PS> $variable = 1
PS> $variable += 'one'
InvalidArgument: Cannot convert value "one" to type "System.Int32". Error: "Input string was not in a
correct format."
```

The `+=` operator may be used to add elements to an existing array:

```
$array = 1, 2
$array += 3
```

You can also use it to add another array:

```
$array = 1, 2
$array += 3, 4
```

The `+=` operator may be used to join two hashtables:

```
$hashtable = @{key1 = 1}
$hashtable += @{key2 = 2}
```

As seen when using the addition operator, the operation fails if one of the keys already exists.

The subtract and assign operator (`-=`) is intended for numeric operations, as shown in the following examples:

```
$i = 20
$i -= 2
```

After this operation has completed, `$i` has a value of `18`.

Multiply and assign, divide and assign, and modulus and assign

Numeric assignments using the multiply and assign operator may be performed using `*=`. The value held by the variable `i` is `4`:

```
$i = 2
$i *= 2
```

The multiply and assign operator may be used to duplicate a string held in a variable:

```
$string = 'one'
$string *= 2
```

The value on the right-hand side of the `*=` operator must be numeric or must be able to convert to a number. For example, a string containing the number `2` is acceptable:

```
$string = 'one'
$string *= '2'
```

Using a string that PowerShell cannot convert to a number will result in an error, as follows:

```
PS> $variable = 'one'
PS> $variable *= 'one'
InvalidArgument: Cannot convert value "one" to type "System.Int32". Error: "Input string was not in a
correct format."
```

The multiply and assign operator may be used to duplicate an array held in a variable. In the following example, the variable holds the value `1, 2, 1, 2` after this operation:

```
$variable = 1, 2
$variable *= 2
```

The assign and divide operator is used to perform numeric operations. The variable holds a value of `1` after the following operation:

```
$variable = 2
$variable /= 2
```

The remainder and assign operator assigns the result of the remainder operation to a variable:

```
$variable = 10
$variable %= 3
```

After the preceding operation, the variable holds a value of `1`, which is the remainder when dividing `10` by `3`.

Statements can be assigned to a variable

In PowerShell, statements can be assigned to variables. All output from that statement will be captured in the variable.

The most common use for this is capturing arrays of results. The following example creates a custom object that includes output from both the `Win32_Service` CIM class and the `Get-Process` command. All of the generated objects are assigned to the `$serviceInfo` variable:

```
$services = Get-CimInstance Win32_Service -Filter 'State="Running"'
$serviceInfo = foreach ($service in $services) {
    $process = Get-Process -ID $service.ProcessID
    [PSCustomObject]@{
        Name        = $service.Name
        ProcessName = $process.Name
        ProcessID   = $service.ProcessID
        Path        = $process.Path
        MemoryUsed  = $process.WorkingSet64 / 1MB
    }
}
```

Assigning statements such as loops in PowerShell is the most efficient way of gathering a set of results into a variable.

Comparison operators

Comparison operators can be used to compare two scalar values. PowerShell has a wide variety of comparison operators, which are as follows:

- **Equal to and not equal to**: `-eq` and `-ne`
- **Like and not like**: `-like` and `-notlike`
- **Greater than and greater than or equal to**: `-gt` and `-ge`
- **Less than and less than or equal to**: `-lt` and `-le`
- **Contains and not contains**: `-contains` and `-notcontains`
- **In and not in**: `-in` and `-notin`

Case sensitivity

None of the comparison operators are case-sensitive by default. Each of the comparison operators has two additional variants, one that explicitly states it is case sensitive, and another that explicitly states it is case insensitive.

For example, the following statement returns `true`:

```
'Trees' -eq 'trees'
```

Adding a `c` prefix in front of the operator name forces PowerShell to make a case-sensitive comparison. The following statement returns `false`:

```
'Trees' -ceq 'trees'
```

In addition to the case-sensitive prefix, PowerShell also has an explicit case-insensitive modifier. In the following example, the statement returns `true`:

```
'Trees' -ieq 'trees'
```

However, as case-insensitive comparison is the default, it is extremely rare to see examples of the `i` prefix.

These behavior prefixes can be applied to all of the comparison operators.

Comparison operators and arrays

When comparison operators are used with scalar values (a single item as opposed to an array), the comparison results in `true` or `false`.

When a comparison operator is used with an array or collection, the result of the comparison is all matching elements. That is,

the array is enumerated and all successfully compared values are returned. For example:

```
1, $null -ne $null                  # Returns 1
1, 2, 3, 4 -ge 3                    # Returns 3, 4
'one', 'two', 'three' -like '*e*'   # Returns one, three
```

This array comparison behavior does not apply when using `-contains`, `-notcontains`, `-in`, or `-notin`.

Comparisons to null

Returning each matching value from an array can be problematic if a comparison is used to test whether a variable holding an array exists.

In the following example, `-eq` is successfully used to test that a value has been assigned to a variable called `array`:

```
$array = 1, 2
if ($array -eq $null) { Write-Host 'Variable not set' }
```

This test is valid as long as the array does not hold two or more null values. When two or more values are present, the condition unexpectedly returns `$true`:

```
PS> $array = 1, 2, $null, $null
PS> if ($array -eq $null) { Write-Host 'No values in array' }
No values in array
```

This happens because the result of the comparison is an array with two null values. PowerShell returns matching values from the array, not just `true` or `false`. If it were a single null value, PowerShell would flatten the array. With two values, PowerShell cannot do that. The effect of two null values can be seen when casting each to `Boolean`:

```
[Boolean]@($null)           # Returns false
[Boolean]@($null, $null)    # Returns true
```

To avoid this problem, `$null` must be on the left-hand side of the expression. For example, the following `Write-Host` statement does not execute; the `array` variable is not null:

```
$array = 1, 2, $null, $null
if ($null -eq $array) { Write-Host 'Variable not set' }
```

In this case, the array is not enumerated; null is compared with the entire array. The result will be `$false`; the `array` variable is set.

Equal to and not equal to

The `-eq` (equal to) and `-ne` (not equal to) operators perform exact (and, by default, case-insensitive) comparisons. In the following example, `$true` is returned:

```
1 -eq 1
'string' -eq 'string'
[char]'a' -eq 'A'
$true -eq 1
$false -eq 0
```

Similarly, `-ne` (not equal) will return `$true` for each of the following:

```
20 -ne 100
'this' -ne 'that'
$false -ne 'false'
```

The last example compares `$false`, the Boolean, with a string containing the word `false`. PowerShell will attempt to convert the word, but as the word is not an empty string, the result will be `$true`.

Like and not like

The `-like` and `-notlike` operators support wildcards. `*` matches a string of any length (zero or more characters long) and `?` matches a single character. Each of the following examples returns `$true`:

```
'The cow jumped over the moon' -like '*moon*'
'Hello world' -like '??llo w*'
'' -like '*'
'' -notlike '?*'
```

Behind the scenes, PowerShell turns expressions used with `-like` and `-notlike` into regular expressions.

Greater than and less than

When comparing numbers, each of the operators `-ge` (greater than or equal to), `-gt` (greater than), `-le` (less than or equal to), and `-lt` (less than) is simple to use:

```
1 -ge 1       # Returns true
2 -gt 1       # Returns true
1.4 -lt 1.9   # Returns true
1.1 -le 1.1   # Returns true
```

String comparison with operators places `0` first, then each lower case and upper case character in turn. For example, from least to greatest, `0123456789aAbBcCdD...xXyYzZ`. Also, it is important to note the following:

- Cultural variants of characters. For example, the character *å* falls between *A* and *b* in the list.
- Other alphabets. For example, Cyrillic and Greek come after the Roman alphabet (after *Z*).

Comparisons can be made in a specific culture when using commands such as `Sort-Object` with the `Culture` parameter. Comparisons are always based on en-US when using the operators:

```
'apples' -lt 'pears'    # Returns true
'Apples' -lt 'pears'    # Returns true
'bears' -gt 'Apples'    # Returns true
'å' -gt 'a'             # Returns true
```

When a case-sensitive operator, such as `-clt`, is used, `B` can be seen to fall between `b` and `c`:

```
'bat' -clt 'Bat'   # True, b before B
'Bat' -clt 'cat'   # True, B before c
```

The use of greater than and less than with strings may often be difficult to apply. Careful testing is recommended.

Contains and in

The `-contains`, `-notcontains`, `-in`, and `-notin` operators are used to test the content of arrays.

When using `-contains` or `-notcontains`, the array must be on the left-hand side of the operator:

```
1, 2 -contains 2       # Returns true
1, 2, 3 -contains 4    # Returns false
```

When using `-in` or `-notin`, the array must be on the right-hand side of the operator:

```
1 -in 1, 2, 3    # Returns true
4 -in 1, 2, 3    # Returns false
```

Contains or in?

When using comparison operators, I tend to write the subject (the item I want to compare) on the left and the object on the right. Comparisons to null are an exception to this rule: null is placed on the left. The subject is the variable or property I am testing; the object is the thing I am testing against. For example, I might set the subject to a user in Active Directory:

```
$subject = Get-ADUser -Identity $env:USERNAME -Properties @(
    'department'
    'memberOf'
)
```

I use `contains`, where the subject is an array, and the object is a single value:

```
$subject.MemberOf -contains 'CN=Group,DC=domain,DC=example'
```

I use `in`, where the subject is a single value, and the object is an array:

```
$subject.Department -in 'Department1', 'Department2'
```

Regular expression-based operators

Regular expressions are an advanced form of pattern matching. In PowerShell, some operators have direct support for regular expressions. Regular expressions themselves are covered in greater detail in *Chapter 9, Regular Expressions*.

The following operators use regular expressions:

- **Match**: `-match`
- **Not match**: `-notmatch`
- **Replace**: `-replace`
- **Split**: `-split`

Match and not match

The `-match` and `-notmatch` operators test whether a string matches a regular expression. If so, the operators will return `$true` or `$false`:

```
'The cow jumped over the moon' -match 'cow'  # Returns true
'The       cow' -match 'The +cow'            # Returns true
```

In the preceding example, the `+` symbol is reserved; it indicates that `The` is followed by one or more spaces before `cow`.

Match is a comparison operator

Like the other comparison operators, if `-match` (or `-notmatch`) is used against an array, it returns each matching element instead of `true` or `false`. The following comparison returns the values `one` and `three`:

```
"one", "two", "three" -match 'e'
```

When match is used against an array, the `$matches` variable is not set.

In addition to returning a `$true` or `$false` value about the state of the match, a successful match adds values to a reserved variable, `$matches`. For example, the following regular expression uses a character class (a set of values enclosed in square brackets) to indicate that it should match any character from `0` to `4`, repeated 0 or more times:

```
'1234567689' -match '[0-4]*'
```

Once the match has been executed, the `$matches` variable (a hashtable) is populated with the part of the string that matched the expression:

```
PS> $matches
Name                           Value
----                           -----
0                              1234
```

Regular expressions use parentheses to denote groups. Groups may be used to capture interesting elements of a string:

```
PS> 'Group one, Group two' -match 'Group (.*), Group (.*)'
True
PS> $matches
Name                  Value
----                  -----
2                     two
1                     one
0                     Group one, Group two
```

The captured value `one` is held in the group named `1`, and is accessible using either of the following statements:

```
$matches[1]
$matches.1
```

The `$matches` variable is an automatically filled hashtable; in the preceding example, the value `1` is being used as a key to access a capture group.

Replace

The `-replace` operator performs string replacement based on a regular expression. For example, it can be used to replace several instances of the same thing:

```
PS> 'abababab' -replace 'a', 'c'
cbcbcbcb
```

In the example, `a` is the regular expression that dictates what must be replaced. `'c'` is the value any matching values should be replaced with.

The syntax for the `-replace` operator can be generalized, as follows:

```
<Value> -replace <Match>, <Replace-With>
```

If the `Replace-With` value is omitted, the matches are replaced with nothing (that is, they are removed):

```
PS> 'abababab' -replace 'a'
bbbb
```

Regular expressions use parentheses to capture groups (a sub-string of the original). The `-replace` operator can use those groups. Each group may be used in the `Replace-With` argument. For example, a set of values can be reversed:

```
'value1,value2,value3' -replace '(.*),(.*),(.*)', '$3,$2,$1'
```

In the preceding regular expression, `.*` matches zero or more of any character. Each capture group is expected to be separated by a comma value.

The values `$1`, `$2`, and `$3` are references to each of the capture groups in the order they appear in the expression. When performing this operation, the `Replace-With` argument uses single quotes to prevent PowerShell from evaluating the group references as if they were variables. This problem is shown in the following example. The first attempt works as expected; the second shows an expanded PowerShell variable instead:

```
PS> 'value1,value2,value3' -replace '(.*),(.*),(.*)', '$3,$2,$1'
value3,value2,value1
PS> $1 = $2 = $3 = 'Oops'
PS> 'value1,value2,value3' -replace '(.*),(.*),(.*)', "$3,$2,$1"
Oops,Oops,Oops
```

Capture groups are explored in greater detail in *Chapter 9, Regular Expressions*.

The `-replace` operator is incredibly useful and widely used.

Split

The `-split` operator splits a string into an array based on a regular expression.

The following example splits a string into an array containing `a`, `b`, `c`, and `d` based on each of the numbers in the string:

```
PS> 'a1b2c3d4' -split '[0-9]'
a
b
c
d
```

The syntax for the `-split` operator can be generalized, as follows, only the `Match` argument is mandatory:

```
<Value> -split <Match>, <Maximum-Number>, <Split-Options>
```

The results of a split can be assigned to one or more variables. For example:

```
$first, $second, $third = '1,2,3,4,5' -split ','
```

The string `1` will be assigned to `$first` and `2` to `$second`. The variable `$third` will contain the strings `3`, `4`, and `5` as an array. Using `$null` instead would discard `3`, `4`, and `5`:

```
 $first, $second, $null = '1,2,3,4,5' -split ','
```

`Maximum-Number` can be used to limit the number of split operations performed on the string on the left-hand side. The default value is `0`, meaning an unlimited number of split operations. The following example will result in an array containing two elements: the first element will be the string `1`, the second element will be `2`, `3`, `4`, `5`:

```
$split = '1,2,3,4,5' -split ',', 2
```

The `Split-Options` field is used to change how the match is performed. It includes a number of options:

- `SimpleMatch`
- `RegexMatch`
- `CultureInvariant`
- `IgnorePatternWhitespace`
- `Multiline`
- `Singleline`
- `IgnoreCase`
- `ExplicitCapture`

Microsoft Docs has descriptions of each of these possible values:

https://docs.microsoft.com/dotnet/api/system.management.automation.splitoptions?view=powershellsdk-7.0.0

For example, the `SimpleMatch` option changes `-split` to be explicit instead of using a regular expression as shown in the following example:

```
'a?b?c?d?' -split 'b?', 0, 'SimpleMatch'
```

The preceding example splits the string in two, the first value will be `a?`, the second `c?d?`. If `SimpleMatch` is taken away the result will be very different. In a regular expression, the `?` character makes the preceding character optional.

Multiple options can be used as a comma-separated list. For example:

```
'axbxcxd' -csplit ' X ', 0, 'IgnoreCase, IgnorePatternWhiteSpace'
```

The `-csplit` variant of the `-split` operator makes the match case sensitive; setting the `IgnoreCase` option switches back to case insensitive.

Logical operators

Logical operators evaluate two or more comparisons or other operations that produce a Boolean (`true` or `false`) result.

The following logic operators are available:

- **And**: `-and`
- **Or**: `-or`
- **Exclusive or**: `-xor`
- **Not**: `-not` and `!`

And

The `-and` operator returns `$true` if the values on the left-hand and right-hand side are both `$true`.

For example, each of the following returns `$true`:

```
$true -and $true
1 -lt 2 -and "string" -like 's*'
1 -eq 1 -and 2 -eq 2 -and 3 -eq 3
(Test-Path C:\Windows) -and (Test-Path 'C:\Program Files')
```

Or

The `-or` operator returns `true` if the value on the left, or the value on the right, or both, are `true`.

For example, each of the following returns `$true`:

```
$true -or $true
2 -gt 1 -or "something" -ne "nothing"
1 -eq 1 -or 2 -eq 1
(Test-Path C:\Windows) -or (Test-Path D:\Windows)
```

Exclusive or

The `-xor` operator will return `$true` if either the value on the left is `$true`, or the value on the right is `true`, but not both.

For example, each of the following returns `$true`:

```
$true -xor $false
1 -le 2 -xor 1 -eq 2
(Test-Path C:\Windows) -xor (Test-Path D:\Windows)
```

The `-xor` operator is perhaps one of the most rarely used in PowerShell.

Not

The `-not` (or `!`) operator may be used to negate the expression that follows it.

For example, each of the following returns `true`:

```
-not $false
-not (Test-Path X:\)
-not ($true -and $false)
!($true -and $false)
```

Double negatives

The `-not` operator has an important place, but it is worth rethinking an expression if it injects a double negative. For example, the following expression returns `true`:

```
-not (1 -ne 1)
```

The preceding expression is better written using the `-eq` operator:

```
1 -eq 1
```

Binary operators

Binary operators are used to perform bitwise operations in PowerShell, that is, operations based around the bits that make up a numeric value. Each operator returns the numeric result of a bitwise operation.

All numeric values can be broken down into bytes and, in turn, bits.

Each byte is made up of 8 bits. Each bit in the byte has a value based on its position, highest value (or most significant) first. These bits can be combined to make up any number between 0 and 255.

The possible bit values for a byte can be represented as a table:

Bit position	1	2	3	4	5	6	7	8
Bit value	128	64	32	16	8	4	2	1

Table: 6.1: Bit values of a byte

The available operators are:

- **Binary and**: `-band`
- **Binary or**: `-bor`
- **Binary exclusive or**: `-bxor`
- **Binary not**: `-bnot`
- **Shift left**: `-shl`
- **Shift right**: `-shr`

Binary and

The result of `-band` is a number where each of the bits in both the value on the left and the value on the right is set.

In the following example, the result is `2`:

```
11 -band 6
```

This bitwise AND operation can be shown in a table. Each value is shown in binary:

Bit value		8	4	2	1
Left-hand side	11	1	0	1	1
Right-hand side	6	0	1	1	0
`-band`	2	0	0	1	0

Table 6.2: 11 -band 6

The result of `-band` is a number where both the value on the left-hand side and the value on the right-hand side include the bit.

Binary or

The result of `-bor` is a number where the bits are set in either the value on the left or right.

In the following example, the result is `15`:

```
11 -bor 12
```

This operation can be shown in a table:

Bit value		8	4	2	1
Left-hand side	11	1	0	1	1
Right-hand side	12	1	1	0	0
`-band`	15	1	1	1	1

Table 6.3: 11 -bor 12

The result is a number made up of the bits from each number where either number has the bit set.

Binary exclusive or

The result of `-bxor` is a number where the bits are set in either the value on the left or the value on the right, but not both.

In the following example, the result is `11`:

```
6 -bxor 13
```

This operation can be shown in a table:

Bit value		8	4	2	1

Left-hand side	6	0	1	1	0
Right-hand side	13	1	1	0	1
`-band`	11	1	0	1	1

Table 6.4: 6 -bxor 13

The `-bxor` operator is useful for toggling bit values. For example, `-bxor` might be used to toggle the `AccountDisable` bit of `UserAccountControl` in Active Directory:

```
512 -bxor 2 # Result is 514 (Disabled, 2 is set)
514 -bxor 2 # Result is 512 (Enabled, 2 is not set)
```

Binary not

The `-bnot` operator is applied before a numeric value; it does not use a value on the left-hand side. The result is a value that's composed of all bits that are not set.

The `-bnot` operator works with signed and unsigned 32-bit and 64-bit integers (`Int32` , `UInt32` , `Int64` , and `UInt64`). When working with 8-bit or 16-bit integers (`SByte` , `Byte` , `Int16` , and `UInt16`), the result is always a signed 32-bit integer (`Int32`).

In the following example, the result is `-123` :

```
-bnot 122
```

As the preceding result is a 32-bit integer (`Int32`), it is difficult to show the effect in a small table. If this value were an `SByte` , the operation could be expressed in a table as follows:

Bit value		Signing	64	32	16	8	4	2	1
Before `-bnot`	122	0	1	1	1	1	0	1	0

After `-bnot`	-123	1	0	0	0	0	1	0	1

Table 6.5: -bnot 122

As shown in the preceding table, the `-bnot` operator reverses the value for each bit. The signing bit is not treated any differently.

Shift left and shift right operators

The `-shl` and `-shr` operators were introduced with PowerShell 3.0. These operators perform bit-shifting.

The `-shl` and `-shr` operators have the lowest precedence and are only executed after all other operators. For example, the result of the following calculation is `128`; the multiplication and addition operators are evaluated before `-shl`.

```
2 * 4 -shl 2 + 2
```

The effect of shift operators is best demonstrated by representing numeric values in binary. For the value of `78`, the following bits must be set:

Bit value	128	64	32	16	8	4	2	1
On or off	0	1	0	0	1	1	1	0

Table 6.6: Bit values

When a left shift operation is performed, every bit is moved a defined number of places to the left; in the following example, one bit to the left:

```
78 -shl 1
```

The result is `156`, which is expressed in this bit table:

Bit value	128	64	32	16	8	4	2	1
Before shift	0	1	0	0	1	1	1	0

After shift	1	0	0	1	1	1	0	0

Table 6.7: Shift left

Shifting one bit to the right reverses the operation:

```
PS> 156 -shr 1
78
```

When shift left (`-shl`) operator converting values using left or right shifting, bits that are set and right-shifted past the rightmost bit (bit value `1`) become `0` . For example:

```
PS> 3 -shr 1
1
```

This is expressed in the following table. Bits that end up in the rightmost column are discarded; they are outside of the range of bits used by the numeric value:

Bit value	128	64	32	16	8	4	2	1	Out of range
Before shift	0	0	0	0	0	0	1	1	
After shift	0	0	0	0	0	0	0	1	1

Table 6.8: Shift right, discarded bits

If the numeric value is of a specific numeric type, the resulting number cannot exceed the maximum value for that type. For example, a `Byte` has a maximum value of `255` ; if the value of `255` is shifted one bit to the left, the resulting value will be `254` :

```
PS> ([Byte]255) -shl 1
254
```

Shifting out of range is shown in this table:

Bit value	Out of range	128	64	32	16	8	4	2	1

Before shift		1	1	1	1	1	1	1	1
After shift	1	1	1	1	1	1	1	1	0

Table 6.9: Shift left, discarded bits

If the value were capable of being larger, such as a 16- or 32-bit integer, the value would be allowed to increase as it no longer falls out of range:

```
PS> [Int16]255 -shl 1
510
```

Bit shifting like this is easiest to demonstrate with unsigned types such as `Byte`, `UInt16`, `UInt32`, and `UInt64`. Unsigned types cannot support values lower than 0 (negative numbers), they have no way of describing a negative value.

Signed types, such as `SByte`, `Int16`, `Int32`, and `Int64`, use the highest-order bit to indicate whether the value is positive or negative. For example, this table shows the bit positions for a signed byte (`SByte`):

Bit position	1	2	3	4	5	6	7	8
Bit value	Signing	64	32	16	8	4	2	1

Table 6.10: Signed byte

The preceding bit values may be used to express numbers between `127` and `-128`. The binary forms of `1` and `-1` are shown as an example in the following table:

Bit value	Signing	64	32	16	8	4	2	1
1	0	0	0	0	0	0	0	1

-1	1	1	1	1	1	1	1	1

Table 6.11: Positive and negative in bits

For a signed type, each bit (except for signing) adds to a minimum value:

- When the signing bit is not set, add each value to `0`
- When the signing bit is set, add each value to `-128`

When applying this to left shift, if the value of `64` is shifted one bit to the left, it becomes `-128`:

```
PS> ([SByte]64) -shl 1
-128
```

The shift left into the signing bit is expressed in the following table:

Bit value	Signing	64	32	16	8	4	2	1
Before shift	0	1	0	0	0	0	0	0
After shift	1	0	0	0	0	0	0	0

Table 6.12: Shift into signing bit

Shift operations such as these are common in the networking world. For example, the IP address `192.168.4.32` may be represented in a number of different ways:

- **In hexadecimal**: `C0A80420`
- **As an unsigned 32-bit integer**: `3232236576`
- **As a signed 32-bit integer**: `-1062730720`

The signed and unsigned versions of an IP address are calculated using left shift. For example, the IP address `192.168.4.32` may be written as a signed 32-bit integer (`Int32`):

```
(192 -shl 24) + (168 -shl 16) + (4 -shl 8) + 32
```

Shift operations such as these can be useful but are not common. The next section explores the assignment operator.

Type operators

Type operators are designed to work with and test .NET types. The following operators are available:

- **As**: `-as`
- **Is**: `-is`
- **Is not**: `-isnot`

These operators may be used to convert an object of one type into another, or to test whether or not an object is of a given type.

As

The `-as` operator is used to attempt to convert a value into an object of a specified type. The operator returns null (without throwing an error) if the conversion cannot be completed.

For example, the operator may be used to perform the following conversions:

```
"1" -as [Int32]
'String' -as [Type]
```

If the attempt to convert the value fails, nothing is returned and no error is raised:

```
$true -as [DateTime]
```

The `-as` operator can be useful for testing whether a value can be cast to a specific type, or whether a specific type exists.

For example, the `System.Windows.Forms` assembly is not imported by default and the `System.Windows.Forms.Form` class does not exist. The `-as` operator may be used to test if it is possible to find the `System.Windows.Forms.Form` type:

```
if (-not ('System.Windows.Forms.Form' -as [Type])) {
    Write-Host 'Adding assembly' -ForegroundColor Green
    Add-Type -Assembly System.Windows.Forms
}
```

If the `System.Windows.Forms` assembly has not been imported, attempting to turn the string, `System.Windows.Forms.Form`, into a type will fail. The failure to convert will not generate an error.

is and isnot

The `-is` and `-isnot` operators test whether a value is of a specified type.

For example, each of the following returns `true`:

```
'string' -is [String]
1 -is [Int32]
[String] -is [Type]
123 -isnot [String]
```

The `-is` and `-isnot` operators are especially useful for testing the exact type of a value, often in an `if` statement when the action taken depends on the value type.

Redirection operators

Chapter 3, Working with Objects in PowerShell, started exploring the different output streams PowerShell utilizes.

Information from a command may be redirected using the redirection operator, `>`. Information may be sent to another stream or a file.

For example, the output from a command can be directed to a file. The file contains the output as it would have been displayed in the console:

```
PS> Get-Process -Id $pid > process.txt
PS> Get-Content process.txt
NPM(K)    PM(M)      WS(M)     CPU(s)      Id  SI ProcessName
 ------    -----      -----     ------      --  -- -----------
     78   144.69     186.91      12.69   15284   1 pwsh
```

Each of the streams in PowerShell has a number associated with it. These are shown in the following table:

Stream name	**Stream number**
Standard out	1
Error	2
Warning	3
Verbose	4
Debug	5
Information	6

Table 6.13: PowerShell streams

Each of the preceding streams can be redirected. In most cases, PowerShell provides parameters for commands, which can be used to capture the streams when used, for example, the `ErrorVariable` and `WarningVariable` parameters.

About Write-Host

Before PowerShell 5, the output written using the `Write-Host` command could not be captured, redirected, or assigned to a variable. In PowerShell 5, `Write-Host` became a wrapper for `Write-Information`; the message is sent to the information stream.

> Information written using `Write-Host` is unaffected by the `InformationPreference` variable and the `InformationAction` parameter, except when either is set to `Ignore`.
>
> When `InformationAction` for the `Write-Host` command is set to `Ignore`, the output will be suppressed. When `Ignore` is set for the `InformationPreference` variable, an error is displayed, stating that it is not supported.

Redirection to a file

Output from a specific stream may be directed by placing the stream number on the left of the redirect operator.

For example, the output written by `Write-Warning` can be directed to a file:

```
function Test-Redirect{
    'This is standard out'
    Write-Warning 'This is a warning'
}
$output = Test-Redirect 3> 'warnings.txt'
```

The `$output` variable will contain the string `This is standard out`. The warning message from stream `3` is redirected to the `warnings.txt` file.

When using the redirect operator, any file of the same name is overwritten. If data is to be appended to a file, the operator is changed to `>>`:

```
function Test-Redirect{
    Write-Warning "Warning $i"
}
$i = 1
Test-Redirect 3> 'warnings.txt'   # Overwrite

$i++
Test-Redirect 3>> 'warnings.txt'  # Append
```

It is possible to redirect additional streams, for example, warnings and errors, by adding more redirect statements. The following example redirects the error and warning streams to separate files:

```
function Test-Redirect{
    'This is standard out'

    Write-Error 'This is an error'
    Write-Warning 'This is a warning'
}
Test-Redirect 3> 'warnings.txt' 2> 'errors.txt'
```

The wildcard character `*` may be used to represent all streams if all content was to be sent to a single file:

```
$verbosePreference = 'continue'
function Test-Redirect {
    'This is standard out'

    Write-Information 'This is information'
    Write-Host 'This is information as well'
    Write-Error 'This is an error'
    Write-Verbose 'This is verbose'
    Write-Warning 'This is a warning'
}
Test-Redirect *> 'alloutput.txt'
```

The preceding example starts by setting the `VerbosePreference` variable. Without this, or the addition of the `verbose`

parameter to the `Write-Verbose` command, the output from `Write-Verbose` will not be shown at all.

> **PowerShell and default file encoding**
>
> The encoding of a file, including text files, can be optionally described using a **BOM** or **Byte-Order Mark**. The BOM is written to the first few bytes of a file and describes the encoding used by the content that follows, that is, how to interpret the bytes in the file to represent characters to display. The BOM is not hidden by most editors.
>
> The different BOM values are described on Wikipedia:
>
> https://wikipedia.org/wiki/Byte_order_mark
>
> The BOM is optional; files without a BOM that are opened in a text editor are generally assumed to be UTF8 (depending on the editor).
>
> In Windows PowerShell, files written using redirection are encoded using UTF-16LE. In PowerShell 7, files are written using UTF8 without a BOM at the beginning of the file.
>
> If PowerShell 7 uses `>>` to append to a file created using `>`, in Windows PowerShell the result will be a file with mixed encoding. The presence of the Unicode BOM renders the content unreadable in most cases.

Streams can be redirected to other streams rather than a file.

Redirecting streams to standard output

Streams can be redirected to standard output in PowerShell. The destination stream is written on the right-hand side of the redirect operator (without a space). Stream numbers on the right-hand side are prefixed with an ampersand (`&`) to distinguish the stream from a filename.

> **Only Stdout**
>
> Each of the following examples shows redirection to `Stdout` , `&1` . It is not possible to redirect to streams other than standard output.

For example, the output from `Write-Information` , stream 6, is redirected to standard output:

```
PS> function Test-Redirect{
>>     'This is standard out'
>>     Write-Information 'This is information'
>> }
PS> $stdOut = Test-Redirect 6>&1
PS> $stdOut
This is standard out
This is information
```

It is possible to redirect additional streams, for example, warnings and errors, by adding more redirect statements. The following example redirects the error and warning streams to standard output:

```
function Test-Redirect {
    'This is standard out'
    Write-Error 'This is an error'
    Write-Warning 'This is a warning'
}
$stdOut = Test-Redirect 2>&1 3>&1
```

The wildcard character `*` may be used to represent all streams if all streams were to be sent to another stream:

```
$verbosePreference = 'continue'
function Test-Redirect {
    'This is standard out'
    Write-Information 'This is information'
    Write-Host 'This is information as well'
    Write-Error 'This is an error'
    Write-Verbose 'This is verbose'
    Write-Warning 'This is a warning'
}
$stdOut = Test-Redirect *>&1
```

The preceding example starts by setting the verbose preference variable. Without this, the output from `Write-Verbose` will not be shown at all.

Redirection to null

Redirecting output to null can be used as a technique used to drop unwanted output. The `$null` variable takes the place of the filename:

```
Get-Process > $null
```

The preceding example redirects standard output (stream 1) to nothing. This is equivalent to using an empty filename:

```
Get-Process > ''
```

Dropping unwanted output is explored further in *Chapter 17, Scripts, Functions, and Script Blocks*.

The stream number or `*` may be included to the left of the redirect operator. For example, warnings and errors might be redirected to `null`:

```
.\somecommand.exe 2> $null 3> $null
.\somecommand.exe *> $null
```

Redirection like this is often used with native executables; redirection is rarely necessary with PowerShell commands.

Other operators

PowerShell has a wide variety of operators, a few of which do not easily fall into a specific category:

- **Call**: `&`
- **Comma**: `,`
- **Format**: `-f`
- **Increment and decrement**: `++` and `--`
- **Join**: `-join`

Each of these operators is in common use. The call operator can run a command based on a string, to the format operator which can be used to build up complex strings, and so on.

Call

The call operator (`&`) is used to execute a string or script block. The call operator is particularly useful when running commands (executables or scripts) that have spaces in the path. The call operator is also useful if the command name changes based on circumstances, for example, the operating system running a command.

The following example runs `pwsh.exe` using a full path held in a string:

```
& 'C:\Program Files\PowerShell\7\pwsh.exe'
```

The path to `pwsh.exe` is normally in the `PATH` environment variable, using the full path as in the example above should not be necessary.

The command itself can also be a variable:

```
$pwsh = 'C:\Program Files\PowerShell\7\pwsh.exe'
& $pwsh
```

This technique can be applied to any command, including PowerShell commands, scripts, and other Windows executables.

Any arguments required by the command can be written in-line, as if the call operator were not present. For example:

```
$pwsh = 'C:\Program Files\PowerShell\7\pwsh.exe'
& $pwsh -NoProfile -NoLogo -Command "Write-Host 'Hello world'"
```

Alternatively, arguments can be supplied as an array, a technique that is useful for commands expecting a large number of arguments:

```
$pwsh = 'C:\Program Files\PowerShell\7\pwsh.exe'
$argumentList = @(
    '-NoProfile'
    '-NoLogo'
    '-Command'
    'Write-Host "Hello world"'
)
& $pwsh $argumentList
```

The call operator may also be used to execute script blocks:

```
$scriptBlock = { Write-Host 'Hello world' }
& $scriptBlock
```

Parameters and arguments can be passed to the script block either in-line, as shown in the previous examples, or using splatting. Splatting is introduced in *Chapter 1, Introduction to PowerShell*.

Comma

The comma operator may be used to separate elements in an array. For example:

```
$array = 1, 2, 3, 4
```

If the comma operator is used before a single value, it creates an array containing one element:

```
$array = ,1
```

When working with functions, the comma operator can be used to emit an array as an object. PowerShell will enumerate an array by default. The `Write-Output` command can be used with the `NoEnumerate` parameter to achieve the same thing.

Format

The `-f` operator can be used to create complex formatted strings. The syntax for the format operator is taken from .NET; Microsoft Docs has a large number of examples:

https://docs.microsoft.com/dotnet/api/system.string.format

The `-f` operator uses a placeholder, a number in curly braces (`{<number>}`) in a string on the left of the operator. The number is the index of a value in an array on the right. For example:

```
PS> '1: {0}, 2: {1}, 3: {2}' -f 'one', 'two', 'three'
1: one, 2: two, 3: three
```

The format operator is one possible way to assemble complex strings in PowerShell. In addition, `-f` may be used to simplify some string operations. For example, a decimal may be formatted as a percentage:

```
'The pass mark is {0:P}' -f 0.8
```

An integer may be formatted as a hexadecimal string:

```
'244 in Hexadecimal is {0:X2}' -f 244
```

A number may be written as a culture-specific currency; in the UK, it will use the £ symbol, in the US, $, and so on:

```
'The price is {0:C2}' -f 199
```

A date may be formatted as a string, which is useful if parts of the date are used in several places in the string:

```
'Today is {0:dddd} the {0:dd} of {0:MMMM}' -f (Get-Date)
```

Reserved characters

When using the `-f` operator, curly braces are considered reserved characters. If a curly brace is to be included in a string as a literal value, it can be escaped:

```
'The value in {{0}} is {0}' -f 1
```

Increment and decrement

The `++` and `--` operators are used to increment and decrement numeric values. The increment and decrement operators are split into pre-increment and post-increment versions.

The post-increment operators are frequently seen in `for` loops. The value for `$i` is used, and then incremented by one after use. In the case of the `for` loop, the value is incremented after the statements inside the loop block have executed:

```
for ($i = 1; $i -le 15; $i++) {
    Write-Host $i -ForegroundColor $i
}
```

The post-decrement reduces the value by one after use:

```
for ($i = 15; $i -ge 0; $i--) {
    Write-Host $i -ForegroundColor $i
}
```

Post-increment and post-decrement operators are often seen when iterating through an array:

```
$array = 1..15
$i = 0
while ($i -lt $array.Count) {
    # $i will increment after this statement has completed.
    Write-Host $array[$i++] -ForegroundColor $i
}
```

Pre-increment and pre-decrement are rarely seen. Instead of incrementing or decrementing a value after use, the change happens before the value is used. For example:

```
$array = 1..5
$i = 0
do {
    # $i is incremented before use, 2 will be the first printed.
    Write-Host $array[++$i]
} while ($i -lt $array.Count -1)
```

The post-increment operator, `++`, is common and is typically used in looping scenarios like those above.

Join

The `-join` operator joins arrays using a string. In the following example, the string is split based on a comma, and then joined based on a tab (`` `t ``):

```
PS> 'a', 'b', 'c', 'd' -join "`t"
a       b       c       d
```

The `-join` operator may also be used in front of an array, when there is no need for a separator. For example:

```
PS> -join ('hello', 'world')
hello world
```

If the parentheses are excluded from the example, the statement will be considered incomplete and will not execute.

Ternary

The ternary operator is a conditional operator that performs a comparison and returns one of two values.

The ternary operator can be used to replace the following statement:

```
$result = if ($value) {
    1
} else {
    2
}
```

When the ternary operator is used, the statement can be simplified to the following:

```
$result = $value ? 1 : 2
```

If `$value` is true, PowerShell sets `$result` to `1`. If `$value` is false, PowerShell sets `$result` to `2`.

As the ternary operator acts on a Boolean value the values 0, null, empty strings, and empty arrays are all considered to be `$false`.

You can use the ternary operator with more complex expressions if the expression is enclosed in parentheses. For example:

```
$value = 1
($value -eq 2) ? 'two' : 'other number'
```

Null coalescing

The null coalescing operator in PowerShell 7 may be used to define a default for a value when the subject is null.

For example, null coalescing is useful if the value for a variable is dependent on another. This operation might be performed using an `if` statement:

```
$valueA = $null
if ($null -eq $valueA) {
    $valueB = 'Default value'
} else {
    $valueB = $valueA
}
```

The null coalescing operator can simplify this expression:

```
$valueA = $null
$valueB = $valueA ?? 'Default value'
```

If `$valueA` is given a non-null value it will be returned as the result of the expression:

```
$valueA = 'Supplied value'
$valueA ?? 'Default value'
```

You can chain ternary operators build up a more complex expression. In the following example, the value of the variable will be set to the first of the functions that returns a non-null value:

```
function first { }
function second { 'second' }
function third { 'third' }
(first) ?? (second) ?? (third)
```

The preceding expression will return the value `'second'` as the function `first` does not return a value. The function `third` will not be called in this case.

If the value were removed from the second function, the result of the expression would be `'third'` as the only non-null value in the expression:

```
function first { }
function second { }
function third { 'third' }
(first) ?? (second) ?? (third)
```

A default value might be added to the end to always ensure the result is never null:

```
(first) ?? (second) ?? (third) ?? 'default'
```

The null coalescing operator allows a complex conditional expression to be defined with a very concise statement. The operator may also be used in an assignment operation.

Null coalescing assignment

The null coalescing assignment operator can be used to simplify the use of the null coalescing operator when assigning values.

In the following example, `$value` will only become `1` if it is null:

```
$value = $null
if ($null -eq $value) {
    $value = 1
}
```

The expression can be simplified by using the null coalescing assignment operator:

```
$value = $null
$value ??= 1
```

In the preceding example, because `$value` is already set, it will not be changed by the second assignment:

```
$value = 1
$value ??= 2
```

The value must be explicitly null (not `false`, or `0`, or an empty string) for the assignment to complete.

This approach might be used to ensure an object exists, for example, a specific process:

```
$process =  Get-Process notepad -ErrorAction SilentlyContinue
$process ??= Start-Process notepad -PassThru
```

In the preceding example, Notepad will only be started if the process does not already exist.

Null conditional

The null conditional operator can be used to avoid errors when a property or method is used on an object, and the object itself is null.

In PowerShell 7.1 and above, the null conditional operators are available as a mainstream feature. In PowerShell 7.0, the null conditional operators are made available by enabling the `PSNullConditionalOperators` experimental feature:

```
Enable-ExperimentalFeature -Name PSNullConditionalOperators
```

Once enabled, and after PowerShell has been restarted, the new operators can be used.

The behavior of the operator is best described by example. The following command will raise an error if the variable has not been set or the value is null:

```
PS> $someObject.ToString()
InvalidOperation: You cannot call a method on a null-valued expression.
```

The null conditional operator can be used so the method is only run if the object is not null:

```
${someObject}?.ToString()
```

As the `?` character can be part of a normal variable name, you must use curly braces to separate the variable name from the null conditional operator.

The same technique can be used for properties of objects that may not be set:

```
$someOtherObject = [PSCustomObject]@{
    Value = $null
}
$someOtherObject.{Value}?.ToString()
```

If a value for the `Value` property is set, the method executes and returns as normal.

This operator cannot be used to avoid errors accessing non-existent properties when strict mode is enabled. See `Get-Help Set-StrictMode` to explore the features and functionality of strict mode in PowerShell.

Pipeline chain

The pipeline chain operators allow conditional execution of commands based on the success (or failure) of another command. Two operators are available, `&&` and `||`.

These operators are present in `cmd.exe` and are also implemented in Bash on Linux.

Evaluating success or failure is based on the value of the `$?` variable.

When the `&&` operator is used, the command on the right-hand side only runs if the command on the left-hand side is successful:

```
PS> function left { 'Doing fine' }
PS> function right { 'Done!' }
PS> left && right
Doing fine
Done!
```

If the left-hand side command fails, the right-hand side command will not run:

```
PS> function left { throw  'Failed' }
PS> function right { 'Done!' }
PS> left && right
Exception: Failed
```

A non-terminating error, such as one raised by `Write-Error`, is not considered a failure by PowerShell when running the function `left`.

A pipeline chain may be useful if installing and then running an application, for example, using the made-up commands shown here:

```
installApplication.exe && application.exe
```

Without `&&`, the `$?` variable, which describes whether the last command succeeded or not, may be used:

```
.\installApplication.exe
if ($?) {
    .\application.exe
}
```

The `||` operator runs the command on the right-hand side even if the left-hand one fails:

```
PS> function left { throw  'Failed' }
PS> function right { 'Done!' }
PS> left || right
Failed
Done!
```

These operators are a small addition to PowerShell and may help users approaching PowerShell from other languages or shells.

Background

The background operator may be used to send the command preceding the operator into a job. For example, running the following command creates a background job:

```
$job = Get-Process &
```

The background job is visible using the `Get-Job` command as shown in the following example:

```
PS> Get-Job
Id   Name  PSJobTypeName  State    HasMoreData     Location   Command
--   ----  -------------  -----    -----------     --------   -------
1    Job1  BackgroundJob  Running  True            localhost  Microsoft.Po...
```

Any output from the job may be retrieved using the `Receive-Job` command:

```
$job = Get-Process &
$job | Receive-Job
```

The job commands are explored in detail in *Chapter 15*, *Asynchronous Processing*.

Summary

This chapter covered many of the operators PowerShell has to offer, including operators for performing arithmetic, assignment, and comparison.

Several specialized operators that use regular expressions were introduced for matching, replacing, and splitting. Regular expressions are explored in *Chapter 9*, *Regular Expressions*. Binary, logical, and type operators were demonstrated.

Finally, several other significant operators were introduced, including call, format, increment, decrement, and the join operator, along with the new ternary, null-coalescing, pipeline chain, and background operators.

In *Chapter 5*, *Variables, Arrays, and Hashtables*, variables, arrays, and Hashtables are explored in detail.

5

Variables, Arrays, and Hashtables

This chapter explores variables, along with a detailed look at arrays and Hashtables, as these have their own complexities.

A variable in a programming language allows you to assign a label to a piece of information or data. A variable can be used and reused in the console, script, or function, or in any other piece of code.

In this chapter, we're going to cover the following topics:

- Naming and creating variables
- Variable commands
- Variable providers
- Scopes and variables
- Types and type conversion
- Typed numeric values
- Objects assigned to variables
- Arrays
- Hashtables
- Lists, dictionaries, queues, and stacks

A variable may be of any .NET type or object instance. The variable may contain a string such as `Hello World`, an integer such as `42`, a decimal such as `3.141`, an array, a Hashtable, a `ScriptBlock`, and so on. Everything a variable might refer to is considered to be an object when used in PowerShell.

Naming and creating variables

Variables in PowerShell are preceded by the dollar symbol (`$`), for example:

```
$MyVariable
```

The name of a variable may contain numbers, letters, and underscores. For example, each of the following is a valid name:

```
$123
$x
$my_variable
$variable
$varIABle
$Path_To_File
```

Variables are frequently written in either camel case or Pascal case. For example:

- `$myVariable` is camel case
- `$MyVariable` is Pascal case

PowerShell does not enforce a naming convention, nor does it consistently use a convention in the automatic variables.

One of the most commonly accepted practices is that variables used as parameters must use Pascal case. Variables used only within a script or a function must use camel case.

I suggest making your variable names meaningful so that when you revisit your script after a long break, you can identify its purpose. I recommend choosing and maintaining a consistent style in your own code.

It is possible to use more complex variable names using the following notation:

```
${My Variable}
${My-Variable}
```

From time to time, the preceding notation appears in PowerShell, perhaps most often in dynamically generated code. This convention is otherwise rare and harder to read and therefore not desirable.

The bracing style has at least one important use. The following example shows an attempt to embed the `var` variable in a string:

```
$var = 'var'
"$variable"   # Will not expand correctly
"${var}iable" # Will expand var
```

The braces define a boundary for the variable name.

The following notation, where a file path is written as the variable name, allows variables to be stored on the filesystem:

```
${C:\Windows\Temp\variable.txt} = "New value"
```

Opening the file shows that the variable value has been set:

```
PS> Get-Content -Path C:\Windows\Temp\variable.txt
New value
```

As with the bracing notation, this is an uncommon practice. It may confuse or surprise anyone reading the code.

Variables do not need to be declared prior to use, nor does a variable need to be assigned a specific type, for example:

```
$itemCount = 7
$dateFormat = "ddMMyyyy"
$numbers = @(1, 9, 5, 2)
$psProcess = Get-Process -Name PowerShell
```

It is possible to assign the same value to several variables in one statement. For example, this creates two variables, `i` and `j`, both with a value of `0`:

```
$i = $j = 0
```

The value types of the values assigned to the variables above are all simple; none are complex objects.

Objects assigned to variables

PowerShell (and all other .NET languages) has two main categories of types, value types and reference types.

A variable that holds a value type contains an instance of that type. Value types include numeric (`int` , `int64` , `byte` , and so on), characters (`char`), `DateTime` , and so on. When a value type is used, for instance, by assigning it to another variable or using the variable as an argument for a parameter, the value is copied.

Reference types differ from value types. A variable assigned a reference type holds only the reference to that object (not the object itself). Two (or more) variables may reference the same object. Reference types are very common in PowerShell; examples include `Hashtable` , `PSCustomObject` , processes returned by `Get-Process` , services returned by `Get-Service` , and so on. When used as an argument for a parameter, the reference to the object is passed (the object is not copied).

A string is not a value type, but it has the same behavior. Strings are immutable – any change to a string will result in the creation of a new string. Strings therefore behave in much the same way as value types.

The following example assigns a single integer value to two variables. Each variable has an independent copy of the integer, despite this being a single statement:

```
$i = $j = 5
```

Changes to the value of one variable have no impact on the value held by the other variable:

```
$i = $j = 5
$i++
$i += 1
$i = $i + 1
```

If each statement is executed in turn, the `$i` variable will be `8`, and the `$j` variable will be `5`.

The following example assigns a single `PSCustomObject` to two variables. Each variable has a reference to the same `PSCustomObject` object:

```
$object1 = $object2 = [PSCustomObject]@{
    Name = 'First object'
}
```

The following statement changes the value of the `Name` property using the `$object1` variable:

```
$object1.Name = 'New name'
```

Because `$object2` holds a reference to the same `PSCustomObject`, the change to the name is also visible in the `$object2` variable:

```
PS> $object1.Name = 'New name'
PS> Write-Host $object2.Name
New name
```

Changes to reference objects apply even if the change is made in a child scope, such as inside a function.

Variable commands

The following commands are used to work with variables:

- `Clear-Variable`
- `Get-Variable`
- `New-Variable`
- `Remove-Variable`
- `Set-Variable`

When using the variable commands, the `$` preceding the variable name is not considered part of the name; `$` tells PowerShell what follows is a variable name.

Clear-Variable

The `Clear-Variable` command removes the value from any existing variable. `Clear-Variable` does not remove the variable itself. For example, the following example calls `Write-Host` twice: on the first occasion, it writes the variable value; on the second occasion, it does not write anything:

```
PS> $temporaryValue = "Some-Value"
PS> Write-Host $temporaryValue -ForegroundColor Green
Some-Value
PS> Clear-Variable temporaryValue
PS> Write-Host $temporaryValue -ForegroundColor Green
```

Get-Variable

The `Get-Variable` command provides access to any variable that has been created in the current session as well as the default (automatic) variables created by PowerShell. For further information on automatic variables, refer to `Get-Help about_Automatic_Variables`.

Default or automatic variables often have descriptions; these may be seen by using the `Get-Variable` command and selecting the description:

```
Get-Variable | Select-Object Name, Description
```

The following example shows the first few variables with descriptions:

```
PS> Get-Variable | Select-Object Name, Description
Name                         Description
----                         -----------
?                            Status of last command
^
$
args
ConfirmPreference            Dictates when confirmation should be requested...
DebugPreference              Dictates the action taken when a Debug message...
EnabledExperimentalFeatures Variable to hold the enabled experimental feat...
Error
ErrorActionPreference        Dictates the action taken when an error messag...
ErrorView                    Dictates the view mode to use when displaying ...
ExecutionContext             The run objects available to cmdlets
false                        Boolean False
FormatEnumerationLimit       Dictates the limit of enumeration on formattin...
HOME                         Folder containing the current user's profile
Host                         A reference to the host of the current runspace
```

Descriptions can be set when creating a variable using the `New-Variable` command.

New-Variable

The `New-Variable` command can be used to create a variable:

```
New-Variable -Name today -Value (Get-Date)
```

The preceding command is the equivalent of using the following assignment:

```
$today = Get-Date
```

`New-Variable` gives more control over the created variable, including adding metadata such as descriptions. For example, it may be used to create a constant, a variable that cannot be changed after creation:

```
New-Variable -Name startTime -Value (Get-Date) -Option Constant
```

Any attempt to modify the variable after creation results in an error message; including changing the variable value or its

properties and attempts to remove the variable, as shown here:

```
PS> New-Variable -Name var -Value 1 -Option Constant
PS> $var = 2
WriteError: Cannot overwrite variable var because it is read-only or constant.
```

A variable cannot be changed into a constant after creation.

Remove-Variable

As the name suggests, the `Remove-Variable` command removes a variable. If this is the only variable referring to an object, the object will be removed from memory shortly afterward.

The `Remove-Variable` command is used as follows:

```
$psProcesses = Get-Process powershell
Remove-Variable psProcesses
```

If more than one variable refers to an object, the object will not be removed, removal only occurs when all references to the object are removed. For example, the following command shows the name of the first process running (`conhost.exe` , in this particular case):

```
PS> $object1 = $object2 = Get-Process | Select-Object -First 1
PS> Remove-Variable object1
PS> Write-Host $object2.Name
conhost
```

Set-Variable

The `Set-Variable` command allows certain properties of an existing variable to be changed. For example, the following sets the value of an existing variable:

```
$objectCount = 23
Set-Variable objectCount -Value 42
```

It is not common to see `Set-Variable` used in this manner; it is simpler to assign the new value directly, as was done when the variable was created.

`Set-Variable` may be used to set a description for a variable:

```
Set-Variable objectCount -Description 'The number of objects in the queue'
```

`Set-Variable` can be used to change a variable scope to private:

```
Set-Variable objectCount -Option Private
```

> **Private scope**
>
> Private scope is accessible using `$private:objectCount` . The `Set` variable may be used but is not required.

The variable commands are one way of interacting with existing variables; the PowerShell provider for variables is another.

Variable provider

PowerShell includes a provider and a drive, which allows variables to be listed, created, and changed using `Get-ChildItem`, `Test-Path`, and so on.

`Get-ChildItem` may be used to list all the variables in the current scope by running the command shown as follows:

```
Get-ChildItem variable:
```

The output will include the built-in variables, as well as any variables created by modules that might have been imported.

As the provider behaves much like a filesystem, `Test-Path` may be used to determine whether a variable exists. `Get-Variable` may be used instead, but `Get-Variable` will throw an error if the variable does not exist:

```
Test-Path variable:\VerbosePreference
```

`Set-Item` may be used to change the value of a variable or create a new variable:

```
Set-Item variable:\new -Value variable
```

`Get-Content` can also be used to retrieve the content of a variable:

```
Get-Content variable:\OutputEncoding
```

The backslash character used in the preceding examples is optional. The output from each `Get-Item` command in the following example is identical:

```
$new = 123
Get-Item variable:\new
Get-Item variable:new
```

Variables are created in a scope, which limits access to a variable and other scoped items.

Scopes and variables

PowerShell uses scopes to limit access to variables (and other items, such as functions). Scopes are layered one on top of another; child scopes inherit from parent scopes. Parent scopes cannot access variables created in child scopes.

There are three named scopes:

- Global
- Script
- Local

Global is the topmost scope; it is the scope the prompt in the console uses and is available to all child scopes.

The Script scope, as the name suggests, is specific to a single script. Script-scoped items are available to all child scopes (such as functions) within that script. The Script scope is also available in modules, making it an ideal place to store variables that should be shared within a module.

Local is the current scope and is therefore relative. In the console, the Local scope is also the Global scope. In a script, the Local scope is the Script scope. Functions and `ScriptBlock` also have a Local scope of their own.

> **More about scopes**
>
> The help document `about_scopes` contains further examples and details.

By default, variables are created in the current scope, the Local scope. However, variables can be accessed from the Local scope

or any parent scope.

Accessing variables

When PowerShell retrieves the value for a variable, it starts by looking for the variable in the Local scope. If the variable does not exist in the Local scope, it looks through parent scopes until it either finds the variable or it runs out of scopes to search.

The following script uses two variables. The variable `$local` exists inside the `Write-VariableValue` function only. The variable `$parent` exists outside, in the parent scope, but can be used inside the function:

```
function Write-VariableValue {
    $local = 'value from inside the function'
    Write-Host "Local: $local"
    Write-Host "Parent: $parent"
}
$parent = 'value from parent scope'
```

Running the function will show both values:

```
PS> Write-VariableValue
Local: value from inside the function
Parent: value from parent scope
```

If the preceding content is run in the console, the parent scope is the Global scope. If the function and the call to the function are inside a script, the parent scope is Script.

If the parent variable were explicitly added to the `Write-VariableValue` function, the locally scoped value would be used instead. For example:

```
function Write-VariableValue {
    $local = 'value from inside the function'
    $parent = 'a new value for parent'
    Write-Host "Local: $local"
    Write-Host "Parent: $parent"
}
$parent = 'value from parent scope'
```

The variable `$parent` inside the function is locally scoped. Assigning a value inside the function has no effect on the value of the variable outside the function. A new variable is created when a value is assigned to a variable for the first time in that scope.

You can use scope modifiers to explicitly define the scope for a variable.

Scope modifiers

A scope modifier is placed before the variable name and is followed by a colon. For example, the following variable will always be created in the Global scope:

```
$Global:variableName = 123
```

Scope modifiers are available for the scopes defined above, Global, Local, and Script. Scope modifiers may also be used for the `private` variable option and provider namespaces such as `alias`, `env`, `function`, and `variable`. The `private` option is explored later in this topic.

Any child scope accessing the variable may repeat the same scope modifier both to make it clear where the variable is from and to avoid using a variable of the same name in the Local (or another parent) scope.

Non-local scoped variables in scripts and functions

> Using user-defined (not automatic) variables from other scopes in scripts and functions can make code very difficult to follow and test. Variable values must be cross-referenced with variable values from elsewhere in a piece of code.
>
> Adding a scope modifier improves the situation as it clearly shows the origin. However, it is still better to avoid out-of-scope variables where possible.

The creation of a child scope is dependent on where the function or script is called, not where it is written.

Numeric scopes

The `Get-Variable` command allows a numeric value to be used for the `Scope` parameter. The numeric value describes how far away from the current scope the variable is. `Get-Variable` may be used when accessing variables in parent scopes, even when a variable of the same name exists in the Local scope.

The following example includes three functions. `first` is called from the topmost scope, then `first` calls `second`, and `second` calls `third`:

```
function first {
    $first = $name = 'first'
    second
}
function second {
    $second = $name = 'second'
    third
}
function third {
    $third = $name = 'third'
}
first
```

Each function has a Local scope, and each function includes a variable called `$name` as well as a variable named after the function. The scopes for these functions are stacked one on top of the other in the order shown in the following list:

1. Global or Script
2. Local scope for `first`
3. Local scope for `second`
4. Local scope for `third`

Each function can access variables defined in the parent scope. That is, the function `third` can access variables defined in the function `second`, variables defined in the function `first`, as well as any variables in the Global or Script scope.

The variables `$first`, `$second`, and `$third` are unique in the script and are therefore accessible in any child scope. The function `third` can access the value of `$first` with no special effort as shown here:

```
function first {
    $first = $name = 'first'
    second
}
function second {
    $second = $name = 'second'
    third
}
function third {
    "The value of first is $first"
}
first
```

Accessing the value of the `$name` variable from a parent scope requires the `Get-Variable` command. As described previously, the `Scope` parameter accepts a numeric value, defined by the number of scopes the given variable is away from the current

scope.

In the function `third`, the value of the variable `$name` in `second` is 1 scope away; the value of the variable `$name` in `first` is 2 scopes away. The changes to the function `third` below show how `Get-Variable` can be used to access those variables:

```
function first {
    $first = $name = 'first'
    second
}
function second {
    $second = $name = 'second'
    third
}
function third {
    "The value of name in first is {0}" -f @(
        Get-Variable -Name name -Scope 2 -ValueOnly
    )
    "The value of name in second is {0}" -f @(
        Get-Variable -Name name -Scope 1 -ValueOnly
    )
}
first
```

In the preceding example, if the function `second` or `third` is run directly, an error will be displayed. For example, running `third` directly will show the following errors:

```
PS> third
Get-Variable:
Line |
   3 |          Get-Variable -Name name -Scope 2 -ValueOnly
     |          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     | The scope number '2' exceeds the number of active scopes. (Parameter 'Scope')
Actual value was 2.
Get-Variable:
Line |
   6 |          Get-Variable -Name name -Scope 1 -ValueOnly
     |          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     | Cannot find a variable with the name 'name'.
```

The technique used in the preceding is instructive but is rarely found in production code.

The ability to access variables from parent scopes is affected by the `private` option, which may be used when creating a variable.

Private variables

A private variable is hidden from child scopes. A private variable may either be created using `New-Variable` or by using the `private` scope modifier as shown in the following examples:

```
New-Variable -Name thisValue -Option Private
$private:thisValue = "Some value"
```

In the following example, a `$name` variable is set in the first two functions. The variable is private in the function `second`.

```
function first {
    $name = 'first'
    second
}
function second {
    $private:name = 'second'
    third
}
function third {
    "The value of name is $name"
}
first
```

When the function `third` uses the `$name` variable, PowerShell searches through parent scopes since the variable does not exist in the local scope. As the variable `$name` is private in the function `second`, that variable value is ignored. PowerShell retrieves the value of the `$name` variable from the scope for the function `first`.

It is still possible to get the value of the private variable using a numeric scope with the `Get-Variable` command as demonstrated in the previous section.

Scopes are an important part of PowerShell and are used by more than variables. Functions, classes, and enums, which are explored in later chapters, are all created in the Local scope and cannot be accessed from a parent scope.

Types and type conversion

Type conversion in PowerShell is used to change between different types of values. Type names are written between square brackets. The type name must be a .NET type, such as a string, an integer (`Int32`), and a date (`DateTime`).

Types may be used to convert, coerce, or cast one type into another. For example, a `DateTime` object returned by `Get-Date` can be cast to a `String` :

```
PS> [String](Get-Date)
10/27/2016 13:14:32
```

Or a string may be changed into a `DateTime` object:

```
PS> [DateTime]"01/01/2016"
01 January 2016 00:00:00
```

If the cast fails, an error will be displayed:

```
[DateTime]'30/30/2016'
InvalidArgument: Cannot convert value "30/30/2016" to type "System.DateTime". Error: "String
'30/30/2016' was not recognized as a valid DateTime."
```

The `-as` operator can be used to cast types as well:

```
'01/01/2016' -as [DateTime]
```

If the cast fails when using `-as` , no error is returned and the result of the expression will be null.

When assigning values, casting may be performed on either the left-hand side or right-hand side of the expression. The following example shows a string cast to XML on the right-hand side of an assignment:

```
$xml = [Xml]'<root><child /></root>'
```

The string value will be coerced into an `XmlDocument` object. The result is shown here:

```
PS> $xml
root
----
root
```

Casting on the right-hand side has no impact on subsequent assignments to the variable.

If the type is placed on the left-hand side of the assignment, the type will also apply to any subsequent assignments to the variable:

```
PS> [String]$thisString = "some value"
PS> $thisString = Get-Date
PS> $thisString.GetType()

IsPublic IsSerial Name                     BaseType
-------- -------- ----                     --------
True     True     String                   System.Object
```

The `Date` is converted to a string because using a type on the left-hand side of `$thisString` when it was created adds an argument-type converter attribute to the variable. The presence of this converter is visible using `Get-Variable`, although viewing the attribute is not particularly useful beyond showing it exists:

```
PS> [String]$thisString = "some value"
PS> (Get-Variable thisString).Attributes
TransformNullOptionalParameters TypeId
------------------------------- ------
                           True ...ArgumentTypeConverterAttribute
```

Getting the type

The target type for a conversion is not visible from the attribute shown in the previous example, but the type can be revealed by using advanced techniques such as Reflection.

The following script shows the type associated with the `$thisString` variable:

```
$attribute = (Get-Variable thisString).Attributes |
    Where-Object TypeId -match 'ArgumentTypeConverterAttribute'
$attribute.GetType().
    GetProperties('Instance,NonPublic').
    GetMethod.Invoke($attribute, @())
```

A variable can only have one type converter attribute at a time. The type converter can be replaced by assigning a new type:

```
PS> [String]$thisString = "some value"
PS> [DateTime]$thisString = Get-Date
PS> $thisString.GetType()
IsPublic IsSerial Name                     BaseType
-------- -------- ----                     --------
True     True     DateTime                 System.ValueType
```

The type conversion applies for the lifetime of the variable: until the session is closed, the variable falls out of scope (when a function or script containing the variable ends), or the variable is removed using `Remove-Variable`.

Setting the variable value to `$null` does not remove the type conversion attribute as shown here:

```
PS> [String]$thisString = 'A string value'
PS> $thisString = $null
PS> $thisString = Get-Process -ID $PID
PS> $thisString.GetType()
IsPublic IsSerial Name                    BaseType
-------- -------- ----                    --------
True     True     String                  System.Object
```

Scripts and typed parameters

Variables with type converters, as shown in the preceding code, can sometimes cause unintended side-effects when writing scripts and functions.

Variables with defined types should not be reused for other purposes in a script or function without considering the assigned type.

Type conversion in PowerShell is exceptionally powerful. A great deal of work is hidden behind a simple cast. The PowerShell team wrote a short blog post listing the steps taken when PowerShell attempts to cast or coerce a value:

https://devblogs.microsoft.com/powershell/understanding-powershells-type-conversion-magic/

PowerShell also allows the use of certain suffix characters when creating numeric types. The most well-known of these are `KB`, `MB`, `GB`, `TB`, and `PB`, which are used to express magnitudes of sizes in bytes. For example:

```
PS> 1GB
1073741824
```

Several other suffixes are available in PowerShell 7.

Typed numeric values

In Windows PowerShell, numbers are automatically created using a type that is able to enclose the value. For example, the value 1 is automatically given the type `Int32`. The value 40,000,000,000, which is too large for `Int32`, is instead created as the type `Int64`.

This automatic assignment continues in PowerShell Core and PowerShell 7, but types may also be explicitly defined without resorting to casting.

For example, the value `1u` has the type `UInt32`; `50l` (that is, 50 followed by the letter *L*) is `Int64`. The change in type can be seen using either `Get-Member` or the `GetType` method. `GetType` is shown here:

```
PS> (50l).GetType()
IsPublic IsSerial Name                    BaseType
-------- -------- ----                    --------
True     True     Int64                   System.ValueType
```

The following list shows the suffixes that can be added to numeric values to influence the resulting type:

- `s`: `short`, `Int16`, or a signed 16-bit integer
- `us`: `ushort`, `UInt16`, or an unsigned 16-bit integer
- `u`: `uint`, `UInt32`, or an unsigned 32-bit integer
- `l`: `long`, `Int64`, or a signed 64-bit integer
- `ul`: `ulong`, `UInt64`, or an unsigned 64-bit integer
- `d`: Decimal

The preceding suffixes are not case-sensitive.

If a value is too large for the requested type, an error will be displayed:

```
PS> 40000000000s
ParserError:
Line |
   1 |  40000000000s
     |             ~
     | At line:1 char:13 + 40000000000s +           ~
The numeric constant 40000000000s is not valid.
```

The value in the preceding snippet, `40000000000`, is too large for the requested short, or 16-bit integer.

Arrays

An array contains a collection of objects. Each entry in the array is called an element, and each element has an index (position). Indexing in an array starts from 0.

Arrays are an important part of PowerShell. When the return from a command is assigned to a variable, an array will be the result if the command returns more than one object. For example, the following command will yield an array of objects:

```
$processes = Get-Process
```

> **Array type**
>
> In PowerShell, arrays are, by default, given the `System.Object[]` type (an array of objects where `[]` is used to signify that it is an array).
>
> Why `System.Object`?
>
> All object instances are derived from a .NET type or class, and, in .NET, every object instance is derived from `System.Object` (including strings and integers). Therefore, a `System.Object` array in PowerShell can hold just about anything.

Arrays in PowerShell (and .NET) are immutable (fixed size). The size is declared on creation and cannot be changed. A new array must be created if an element is to be added or removed. The array operations described next are considered less efficient for large arrays because of the recreation overhead involved in changing the array size.

The following sections explore creating arrays, assigning a type to the array, and selecting elements, as well as adding and removing elements.

Creating an array

There are several ways to create arrays. An empty array (containing no elements) can be created as follows:

```
$myArray = @()
```

An empty array of a specific size may be created using the `New` object. Using `[]` after the name of the type denotes that it is an array, and the number following sets the array size:

```
$myArray = [Object[]]::new(10)        # 10 objects
$byteArray = [Byte[]]::new(100)       # 100 bytes
$ipAddresses = [IPAddress[]]::new(5)  # 5 IP addresses
```

An array with a few strings in it can be created as follows:

```
$myGreetings = "Hello world", "Hello sun", "Hello moon"
```

Or it can be created as follows:

```
$myGreetings = @("Hello world", "Hello sun", "Hello moon")
```

An array may be spread over multiple lines in either the console or a script, which may make it easier to read in a script:

```
$myGreetings = @(
    "Hello world"
    "Hello sun"
    "Hello moon"
)
```

Elements with different types can be mixed in a single array:

```
$myThings = "Hello world", 2, 34.23, (Get-Date)
```

Arrays with a type

An array may be given a type in a similar manner to a variable holding a single value. The difference is that the type name is followed by `[]`, as was the case when creating an empty array of a specific size. For example, each of these is an array type that may appear before a variable name:

```
[String[]]      # An array of strings
[UInt64[]]      # An array of unsigned 64-bit integers
[Xml[]]         # An array of XML documents
```

If a type is set for the array, more care must be taken as regards assigning values. If a type is declared, PowerShell attempts to convert any value assigned to an array element into that type.

In this example, `$null` will become `0` and `3.45` (a double) will become `3` (normal rounding rules apply when converting integers):

```
[Int32[]]$myNumbers = 1, 2, $null, 3.45
```

The following example shows an error being thrown, as a string cannot be converted into an integer:

```
PS> [Int32[]]$myNumbers = 1, 2, $null, "A string"
MetadataError: Cannot convert value "A string" to type "System.Int32". Error: "Input string was not in
a correct format."
```

Adding elements to an array

A single item can be added to the end of an array using the assignment by addition operator:

```
$myArray = @()
$myArray += "New value"
```

The preceding command is equivalent to the following:

```
$myArray = $myArray + "New value"
```

In the background, a new array is created with one extra element. The content of the existing array is copied, then the value for the new element is assigned. Once complete, the original array, stored in memory, is removed. The larger the array, the less efficient this operation becomes.

It is generally better practice to avoid explicitly declaring arrays at all; PowerShell can do the work required to create an array. All statements in PowerShell can be assigned, including loops. For example, the following is a common practice:

```
$array = @()
foreach ($value in 1..5) {
    $array += [PSCustomObject]@{
        Value = $value
    }
}
```

A simpler and more efficient way to write the same statement is shown here:

```
$array = foreach ($value in 1..5) {
    [PSCustomObject]@{
        Value = $value
    }
}
```

If an array is necessary, and the size of the array might change, a `List` (`System.Collections.Generic.List`) is often a better choice. For example, this might be the case if a single loop returns data intended for more than one array. Lists are explored towards the end of this chapter.

The addition operator can be used to join one array to another, demonstrated as follows:

```
$firstArray = 1, 2, 3
$secondArray = 4, 5, 6
$mergedArray = $firstArray + $secondArray
```

This addition operation can also be written as follows:

```
$mergedArray = @(
    $firstArray
    $secondArray
)
```

Using `@()` around a set of elements or expressions is often the cleanest way to merge values into a single array:

```
$firstArray = 1, 2, 3
$mergedArray = @(
    Get-Process
    'someString'
    $firstArray
)
```

Selecting elements from an array

Individual elements from an array may be selected by index. The first and second elements are available using index `0` and `1`:

```
$myArray = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
$myArray[0]
$myArray[1]
```

In a similar manner, array elements can be accessed counting backward from the end. The last element is available using `-1` as the index, and the penultimate element using `-2` as the index. For example:

```
$myArray[-1]
$myArray[-2]
```

Ranges of elements may be selected either going forward (starting from `0`) or going backward (starting with `-1`):

```
$myArray[2..4]
$myArray[-1..-5]
```

More than one range can be selected in a single statement:

```
$myArray[0..2 + 6..8 + -1]
```

This requires some care. The first part of the index set must be an array for the addition operation to succeed; an array cannot be added to an integer. The expression in square brackets is evaluated first and converted into a single array (of indexes) before any elements are selected from the array:

```
PS> $myArray[0 + 6..8 + -1]
InvalidOperation: Method invocation failed because [System.Object[]] does not contain a method named
'op_Addition'.
```

The same error would be shown when running the expression within square brackets alone:

```
0..2 + 6..8 + -1
```

The following modified command shows three different ways to achieve the intended result:

```
$myArray[@(0) + 6..8 + -1]
$myArray[0..0 + 6..8 + -1]
$myArray[@(0; 6..8; -1)]
```

As well as selecting elements by index, elements may be selected by value in several different ways.

The `IndexOf` method may be used to find the position of a value within the array, then the value may be accessed:

```
$index = $myArray.IndexOf(5)
$myArray[$index]
```

If the value of `$index` is `-1`, the value does not exist within the array.

Comparison operators may be used to select elements from the array:

```
$myArray -gt 5
$myArray -lt 3
```

Comparisons can be chained together to build more complex expressions provided each part of the expression continues to return an array:

```
$myArray -gt 3 -lt 7
```

For more complex filtering expressions, `Where-Object` may be used:

```
$myArray | Where-Object { $_ -gt 3 -and $_ -lt 7 }
```

The `Where` method may also be used to filter the array:

```
$myArray.Where{ $_ -gt 3 -and $_ -lt 7 }
```

The `Where` method is faster than `Where-Object` but will raise an error if the variable is not an array.

Changing element values in an array

Elements within an array may be changed by assigning a new value to a specific index, for example:

```
$myArray = 1, 2, 9, 4, 5
$myArray[2] = 3
```

Elements in an array may be changed within a loop. The following example sets all elements in the array to `9`:

```
$myArray = 1, 2, 3, 4, 5
for ($i = 0; $i -lt $myArray.Count; $i++) {
     $myArray[$i] = 9
}
```

Removing elements

Elements cannot be removed from an array because arrays are immutable. To remove an element, a new array must be created.

It is possible to appear to remove an element by setting it to null, for example:

```
$myArray = 1, 2, 3, 4, 5
$myArray[1] = $null
$myArray
```

However, observe that the count or size of the array does not decrease when a value is set to null:

```
PS> $myArray.Count
5
```

Loops (or pipelines) working on the array will not skip the element with the null value (extra code is needed to guard against the null value):

```
$myArray | ForEach-Object { Write-Host $_ }
```

The `Where-Object` command may be used to remove the null value, creating a new array:

```
$myArray | Where-Object { $_ } | ForEach-Object { Write-Host $_ }
```

Depending on usage, several ways are available to address removal.

Removing elements by index

Removing elements based on an index requires the creation of a new array and the omission of the value in the element in that index. In each of the following cases, an array with 100 elements will be used as an example; the element at index 49 (with the value of 50) will be removed:

```
$oldArray = 1..100
```

The following example uses indexes to access and add everything we want to keep:

```
$newArray = $oldArray[0..48] + $oldArray[50..99]
```

Using the .NET `Array.Copy` static method (see *Chapter 7, Working with .NET*), we have the following:

```
$newArray = [Object[]]::new($oldArray.Count - 1)
# Before the index
[Array]::Copy(
    $oldArray,     # Source
    $newArray,     # Destination
    49             # Number of elements to copy
)
# After the index
[Array]::Copy(
    $oldArray,     # Source
    50,            # Copy from index of Source
    $newArray,     # Destination
    49,            # Copy to index of Destination
    50             # Number of elements to copy
)
```

This is the outcome using a `for` loop:

```
$newArray = for ($i = 0; $i -lt $oldArray.Count; $i++) {
    if ($i -ne 49) {
        $oldArray[$i]
    }
}
```

Removing elements by value

Removing an element with a specific value from an array can be achieved in several different ways.

Starting with an array of `100` elements:

```
$oldArray = 1..100
```

Comparison operators can be used to create a new array without certain elements:

```
$newArray = $oldArray -ne 50
```

The `Where-Object` command may be used to identify and omit elements, such as the element with the value `50`. If `50` were to occur more than once, all instances would be omitted:

```
$newArray = $oldArray | Where-Object { $_ -ne 50 }
```

The index of the element might be identified and removed using the methods explored in removing elements according to the index:

```
$index = $oldArray.IndexOf(50)
```

If the value of `$index` returned by the `IndexOf` method is `-1`, the value is not present in the array (`0` would indicate that it is the first element):

```
$index = $oldArray.IndexOf(50)
if ($index -gt -1) {
    $newArray = $oldArray[0..($index - 1)] +
        $oldArray[($index + 1)..99]
}
```

Clearing an array

Finally, an array may be completely emptied by calling the `Clear` method:

```
$newArray = 1, 2, 3, 4, 5
$newArray.Clear()
```

Filling variables from arrays

It is possible to create two (or more) variables when assigning an array:

```
$i, $j = 1, 2
```

Assigning variables from an array can be useful when splitting a string:

```
$firstName, $lastName = "First Last" -split " "
$firstName, $lastName = "First Last".Split(" ")
```

If the array is longer than the number of variables, all remaining elements are assigned to the last variable. For example, the `k` variable will contain `3`, `4`, and `5`, as can be seen as follows:

```
$i, $j, $k = 1, 2, 3, 4, 5
```

If there are too few elements, the remaining variables will not be assigned a value. In this example, `k` will be null:

```
$i, $j, $k = 1, 2
```

Multi-dimensional and jagged arrays

Given that an array contains objects, an array can therefore also contain other arrays.

For example, an array that contains other arrays (a multi-dimensional array) might be created as follows:

```
$arrayOfArrays = @(
    @(1, 2, 3),
    @(4, 5, 6),
    @(7, 8, 9)
)
```

Be careful to ensure that the comma following each of the nested arrays (except the last) is in place. If that comma is missing, the entire structure will be flattened, merging the three inner arrays.

Elements in the preceding array are accessed by indexing into each array in turn (starting with the outermost). The element with the value `2` is accessible using the following notation:

```
PS> $arrayOfArrays[0][1]
2
```

This states that we wish to retrieve the first element (which is an array) and the second element of that array.

The element with the value `6` is accessible using the following:

```
PS> $arrayOfArrays[1][2]
6
```

Jagged arrays are a specific form of multi-dimensional array. An example of a jagged array is as follows:

```
$arrayOfArrays = @(
    @(1,  2),
    @(4,  5,  6,  7,  8,  9),
    @(10, 11, 12)
)
```

As in the first example, it is an array containing arrays. Instead of containing inner arrays, which all share the same size (dimension), the inner arrays have no consistent size (hence, they are jagged).

In this example, the element with the value `9` is accessed as follows:

```
PS> $arrayOfArrays[1][5]
9
```

Multi-dimensional arrays are rarely used in PowerShell. PowerShell tends to flatten arrays, which can make working with complex array structures difficult.

Hashtables

A Hashtable is an associative array or an indexed array. Values in the Hashtable are added with a unique key. Each key has a value associated with it; this is also known as a key-value pair. Keys cannot be duplicated within the Hashtable.

Hashtables are important in PowerShell. They are used to create custom objects, to pass parameters into commands, to create custom properties using `Select-Object`, and as the type for values assigned to parameter values of many different commands, among other things.

The following command may be used to find commands that accept a Hashtable as a parameter type:

```
Get-Command -ParameterType Hashtable
```

This topic explores creating a Hashtable, selecting elements, enumerating all values in a Hashtable, and adding and removing elements.

Creating a Hashtable

An empty Hashtable is created in the same manner as the following:

```
$hashtable = @{}
```

Alternatively, a Hashtable may be created with specific keys and values:

```
$hashtable = @{Key1 = "Value1"; Key2 = "Value2"}
```

Elements in a Hashtable may be spread across multiple lines:

```
$hashtable = @{
    Key1 = "Value1"
    Key2 = "Value2"
}
```

Adding and changing Hashtable elements

Elements may be explicitly added to a Hashtable using the `Add` method:

```
$hashtable = @{}
$hashtable.Add("Key1", "Value1")
```

If the value already exists, using `Add` generates an error (as shown here):

```
PS> $hashtable = @{ Existing = "Value0" }
PS> $hashtable.Add("Existing", "Value1")
MethodInvocationException: Exception calling "Add" with "2" argument(s): "Item has already been added.
Key in dictionary: 'Existing'  Key being added: 'Existing'"
```

The `Contains` method returns true or false, depending on whether a key is present in `$hashtable`. This may be used to test for a key before adding:

```
$hashtable = @{}
if (-not $hashtable.Contains("Key1")) {
    $hashtable.Add("Key1", "Value1")
}
```

Alternatively, two different ways of adding or changing elements are available. The first option uses the key in square brackets:

```
$hashtable = @{ Existing = "Old" }
$hashtable["New"] = "New"              # Add this
$hashtable["Existing"] = "Updated"     # Update this
```

The second option uses a period to access the key as if it were a property:

```
$hashtable = @{ Existing = "Old" }
$hashtable.New = "New"                 # Add this
$hashtable.Existing = "Updated"        # Update this
```

If a value only must be changed if it exists, the `Contains` method may be used:

```
$hashtable = @{ Existing = "Old" }
 if ($hashtable.Contains("Existing")) {
     $hashtable.Existing = "New"
}
```

You can use the `Contains` method to ensure a key is only added if it does not exist:

```
$hashtable = @{ Existing = "Old" }
if (-not $hashtable.Contains("New")) {
     $hashtable.New = "New"
}
```

Keys cannot be added nor can values be changed while looping through the keys in a Hashtable using the `keys` property. Doing so changes the underlying structure of the Hashtable, invalidating the iterator used by the loop operation:

```
$hashtable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
}
foreach ($key in $hashtable.Keys) {
    $hashtable[$key] = "NewValue"
}
OperationStopped: Collection was modified; enumeration operation may not execute.
```

It is possible to work around this problem by first creating an array of the keys in the Hashtable, as follows:

```
$hashtable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
}
[Object[]]$keys = $hashtable.Keys
foreach ($key in $keys) {
    $hashtable[$key] = "NewValue"
}
```

Notice that the highlighted `keys` variable is declared as an array of objects. Using the `Object[]` type conversion forces the creation of a new array object based on the values held in `KeyCollection`. Without this step, the preceding error message would repeat.

Another approach uses the `ForEach-Object` to create a new array of the keys:

```
$hashtable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
}
$keys = $hashtable.Keys | ForEach-Object { $_ }
foreach ($key in $keys) {
    $hashtable[$key] = "NewValue"
}
```

Selecting elements from a Hashtable

Individual elements may be selected by key. Two different formats are supported for selecting elements. Square brackets may be used around the key:

```
$hashtable["Key1"]
```

Or the key name may be used after a period:

```
$hashtable.Key1
```

The key is not case-sensitive, but it is type-sensitive and does not automatically convert. For instance, consider the following Hashtable:

```
$hashtable = @{1 = 'one'}
```

The value `1` can be selected if an integer is used as the key, but not if a string is used. In other words, the following works:

```
$hashtable.1
$hashtable[1]
```

Attempting to use a string to access the value for the key, however, does not work as the key is not a string type:

```
$hashtable."1"
$hashtable["1"]
```

Enumerating a Hashtable

A Hashtable can show the information it holds in several ways. Start with the Hashtable:

```
$hashtable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
}
```

Keys can be returned using the `Keys` property of the Hashtable, which returns `KeyCollection`:

```
$hashtable.Keys
```

Values can be returned using the `Values` property, which returns `ValueCollection`. The key is discarded when using the `Values` property:

```
$hashtable.Values
```

A simple loop can be used to retain the association between `key` and `value`:

```
foreach ($key in $hashtable.Keys) {
    Write-Host "Key: $key    Value: $($hashtable[$key])"
}
```

Alternatively, you can use the `GetEnumerator` method to return an array of objects with key and value properties:

```
PS> $hashtable.GetEnumerator()
Name                           Value
----                           -----
Key2                           Value2
Key1                           Value1
```

Removing elements from a Hashtable

Unlike arrays, removing an element from a Hashtable is straightforward. An element can be removed using the `Remove` method:

```
$hashtable = @{ Existing = "Existing" }
$hashtable.Remove("Existing")
```

If the requested key does not exist, the command does nothing (and does not throw an error).

The `Remove` method cannot be used to modify the Hashtable while looping through the keys in a Hashtable using the `Keys` property:

```
PS> $hashtable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
}
PS> foreach ($key in $hashtable.Keys) {
    $hashtable.Remove($key)
}
OperationStopped: Collection was modified; enumeration operation may not execute.
```

You can use the same method mentioned in the *Adding and changing Hashtable elements* section.

Finally, you can empty a Hashtable by calling the `Clear` method:

```
$hashtable = @{one = 1; two = 2; three = 3}
$hashtable.Clear()
```

Lists, dictionaries, queues, and stacks

Arrays and Hashtables are integral to PowerShell and being able to manipulate these is critical. If these simple structures fail to provide an efficient means of working with a set of data, there are advanced alternatives.

The following .NET collections are discussed:

- `System.Collections.Generic.List`
- `System.Collections.Generic.Dictionary`
- `System.Collections.Generic.Queue`
- `System.Collections.Generic.Stack`

Each of these collections has detailed documentation (for .NET) available on Microsoft Docs:

https://docs.microsoft.com/dotnet/api/system.collections.generic

System.Collections.Generic.List

A list is similar to an array, but with a larger set of features, such as the ability to add elements without copying into a new array. A generic list using the .NET class `System.Collections.Generic.List` is shown next.

`ArrayList` is often used in examples requiring advanced array manipulation in PowerShell. However, `ArrayList` is older (.NET 2.0) and less efficient (it can use more memory), and cannot be strongly typed, as will be shown when creating a generic list. PowerShell does still make use of `ArrayList` internally.

Creating a list

A generic list must have a type declared. A generic list, in this case a list of strings, is created as follows:

```
$list = [System.Collections.Generic.List[String]]::new()
```

`ArrayList` is created in a similar manner. `ArrayList` cannot have the type declared:

```
$arrayList = [System.Collections.ArrayList]::new()
```

Once created, `ArrayList` may be used in much the same way as a generic list.

Adding elements to the list

Elements can be added to a list using the `Add`, `AddRange`, `Insert`, and `InsertRange` methods.

`Add` can be used to add new elements to the end of the list:

```
$list = [System.Collections.Generic.List[String]]::new()
$list.Add("David")
```

`AddRange` can be used to add an array of new elements to the end of the list:

```
$list.AddRange(@("Tim", "Robert"))
```

The `Insert` and `InsertRange` methods are available to add items elsewhere in the list. For example, an element may be added at the beginning:

```
$list.Insert(0, "Sarah")
$list.Insert(2, "Jane")
```

Selecting elements from the list

As with the array, elements may be selected by index:

```
$list = [System.Collections.Generic.List[String]]::new()
$list.AddRange([String[]]("Tom", "Richard", "Harry"))
$list[1]    # Returns Richard
```

The generic list offers a variety of methods that you can use to find elements when the index is not known, such as the following:

```
$index = $list.FindIndex( { $args[0] -eq 'Richard' } )
```

Predicates

In the preceding example, `ScriptBlock` is a predicate. Arguments are passed into `ScriptBlock` and all list items matching the query are returned.

The predicate is similar in syntax to `Where-Object`, except `$args[0]` is used to refer to the item in the list instead of the pipeline variable, `$_`.

A param block may be declared for `ScriptBlock` to assign a more meaningful name to the argument (`$args[0]`) if desirable.

Alternatively, you can use the `IndexOf` and `LastIndex` methods to find elements. Both methods support additional arguments (as opposed to `Array.IndexOf`, which only supports a restrictive search for a value) to constrain the search. For example, the search may start at a specific index:

```
$list.IndexOf('Harry', 2)          # Start at index 2
$list.IndexOf('Richard', 1, 2)     # Start at index 1, and 2 elements
```

Finally, a generic list offers a `BinarySearch` (half-interval) search method. This method may dramatically cut the time to search very large, sorted, datasets when compared to a linear search.

In a binary search, the element in the middle of the list is selected first, and compared to the value. If the value is larger, the first half of the list is discarded, and the element in the middle of the new, smaller, set is selected for comparison. This process repeats (always cutting the list in half) until the value is found (or it runs out of elements to test).

The `Measure-Command` command may be used to time the two searches for a number of different values:

```
$list = [System.Collections.Generic.List[Int]]@(1..100000000)
[PSCustomObject]@{
    # Linear and Binary are roughly comparable
    'Linear, near start' = Measure-Command {
        $list.IndexOf(24)
    } | ForEach-Object TotalMilliseconds
    'Binary, near start' = Measure-Command {
        $list.BinarySearch(24)
    } | ForEach-Object TotalMilliseconds
    # Binary is more effective
    'Linear, near end'   = Measure-Command {
        $list.IndexOf(99767859)
    } | ForEach-Object TotalMilliseconds
    'Binary, near end'   = Measure-Command {
        $list.BinarySearch(99767859)
    } | ForEach-Object TotalMilliseconds
}
```

The time required will vary between different computers. An example of the result is shown below:

```
Linear, near start Binary, near start Linear, near end Binary, near end
------------------ ------------------ ---------------- ----------------
            1.5307             1.4244          44.3841            2.186
```

The time taken to execute a binary search remains roughly constant, regardless of the element position. The time taken to execute a linear search increases as every element must be read (in sequence).

Removing elements from the list

You can remove elements from a list based on the index or value as shown in the following examples:

```
$list = [System.Collections.Generic.List[String]]::new()
$list.AddRange([String[]]("Tom", "Richard", "Harry", "David"))
$list.RemoveAt(1)          # By Richard by index
$list.Remove("Richard")    # By Richard by value
```

All instances of a value may be removed using the `RemoveAll` method:

```
$list.RemoveAll( { $args[0] -eq "David" } )
```

Changing element values in a list

Elements within a list may be changed by assigning a new value to a specific index, as in the following example:

```
$list = [System.Collections.Generic.List[Int]]::new()
$list.AddRange([Int[]](1, 2, 2, 4))
$list[2] = 3
```

System.Collections.Generic.Dictionary

A dictionary, specifically the .NET class `System.Collections.Generic.Dictionary`, is similar to a Hashtable. Like a Hashtable, it is a form of associative array.

Unlike the Hashtable, however, a dictionary implements a type for both the key and the value, which may make it easier to use.

Creating a dictionary

A dictionary must declare a type for the key and value when it is created. You can create a dictionary that uses a string for the key and an IP address for the value using any of the following examples:

```
$dictionary = New-Object System.Collections.Generic.Dictionary"[String,IPAddress]"
$dictionary = New-Object "System.Collections.Generic.Dictionary[String,IPAddress]"
$dictionary = [System.Collections.Generic.Dictionary[String,IPAddress]]::new()
```

Adding and changing elements in a dictionary

As with the Hashtable, the `Add` method may be used to add a new value to a dictionary:

```
$dictionary.Add("Computer1", "192.168.10.222")
```

If the key already exists, using `Add` will generate an error, as was the case with the Hashtable.

In a dictionary, the `Contains` method does not work in PowerShell. When checking for the existence of a key, the `ContainsKey` method should be used as follows:

```
if (-not $dictionary.ContainsKey("Computer2")) {
    $dictionary.Add("Computer2", "192.168.10.13")
}
```

The dictionary supports the addition of elements using dot-notation:

```
$dictionary.Computer3 = "192.168.10.134"
```

The dictionary leverages PowerShell's type conversion for both the key and the value. For example, if a numeric key is used, it is converted into a string. If an IP address is expressed as a string, it will be converted into an `IPAddress` object.

For example, consider the addition of the following element:

```
$dictionary.Add(1, 20)
```

In this case, key `1` is converted into a string, and the value `20` is converted into an IP address. Inspecting the element afterward shows the following:

```
PS> $dictionary."1"
Address            : 20
AddressFamily      : InterNetwork
ScopeId            :
IsIPv6Multicast    : False
IsIPv6LinkLocal    : False
IsIPv6SiteLocal    : False
IsIPv6Teredo       : False
IsIPv4MappedToIPv6 : False
IPAddressToString  : 20.0.0.0
```

Selecting elements from a dictionary

Individual elements may be selected by a key. As with the Hashtable, two different notations are supported:

```
$dictionary["Computer1"]    # Key reference
$dictionary.Computer1       # Dot-notation
```

When using a Hashtable, the type of the key must match the type used when creating the key. With a dictionary, because the key is strongly typed, PowerShell will handle the conversion to an appropriate type.

The following example uses a number as a string for the key:

```
$dictionary = [System.Collections.Generic.Dictionary[String,IPAddress]]::new()
$dictionary.Add("1", "192.168.10.222")
$dictionary.Add("2", "192.168.10.13")
```

Each of the following examples can be used to access the value:

```
$dictionary."1"
$dictionary[1]
$dictionary["1"]
```

Enumerating a dictionary

You can view the information a dictionary holds in several ways. Start with this dictionary:

```
$dictionary = [System.Collections.Generic.Dictionary[String,IPAddress]]::new()
$dictionary.Add("Computer1", "192.168.10.222")
$dictionary.Add("Computer2", "192.168.10.13")
```

Keys can be returned using the `Keys` property of the dictionary, which returns `KeyCollection`:

```
$dictionary.Keys
```

Values can be returned using the `Values` property, which returns `ValueCollection`. The key is discarded when using the `Values` property:

```
$dictionary.Values
```

A simple loop can be used to retain the association between `key` and `value`:

```
foreach ($key in $dictionary.Keys) {
    Write-Host "Key: $key    Value: $($dictionary[$key])"
}
```

Removing elements from a dictionary

An element may be removed from a dictionary using the `Remove` method:

```
$dictionary.Remove("Computer1")
```

The `Remove` method cannot be used to modify the dictionary while looping through the keys in a dictionary using the `Keys` property.

System.Collections.Generic.Queue

A queue is a first-in, first-out array. Elements are added to the end of the queue and taken from the beginning.

The queue uses the .NET class, `System.Collections.Generic.Queue`, and must have a type set.

Creating a queue

A queue of strings may be created as follows:

```
$queue = [System.Collections.Generic.Queue[String]]::new()
```

Enumerating the queue

PowerShell displays the contents of a queue in the same way as it would the contents of an array. It is not possible to access elements of the queue by the index. The `ToArray` method may be used to convert the queue into an array if required:

```
$queue.ToArray()
```

The preceding command returns an array of the same type as the queue. That is, if the queue is configured to hold strings, the array will be an array of strings.

The queue has a `Peek` method that allows the retrieval of the next element in the queue without it being removed:

```
$queue.Peek()
```

The `Peek` method throws an error if the queue is empty (refer to the *Removing elements from the queue* section).

Adding elements to the queue

Elements are added to the end of the queue using the `Enqueue` method:

```
$queue.Enqueue("Tom")
$queue.Enqueue("Richard")
$queue.Enqueue("Harry")
```

Removing elements from the queue

Elements are removed from the start of the queue using the `Dequeue` method:

```
$queue.Dequeue() # This returns Tom.
```

If the queue is empty and the `Dequeue` method is called, an error is thrown, as shown here:

```
PS> $queue.Dequeue()
MethodInvocationException: Exception calling "Dequeue" with "0" argument(s): "Queue empty."
```

To avoid this, the `Count` property of the queue may be inspected, for example:

```
# Set-up the queue
$queue = New-Object System.Collections.Generic.Queue[String]
"Tom", "Richard", "Harry" | ForEach-Object {
    $queue.Enqueue($_)
}
# Dequeue until the queue is empty
while ($queue.Count -gt 0) {
    Write-Host $queue.Dequeue()
}
```

System.Collections.Generic.Stack

A stack is a collection of objects in which objects are accessed in **Last-In, First-Out** (**LIFO**) order. Elements are added and removed from the top of the stack.

The stack uses the .NET class `System.Collections.Generic.Stack` and must define the type of the objects it contains when it is created.

Stacks in PowerShell are used by commands such as `Push-Location` and `Pop-Location`.

Creating a stack

A stack containing strings may be created as follows:

```
$stack = [System.Collections.Generic.Stack[String]]::new()
```

Enumerating the stack

PowerShell displays the content of a stack in the same way as it would the content of an array. It isn't possible to index into a stack. The `ToArray` method may be used to convert the stack into an array if required:

```
$stack.ToArray()
```

The preceding command returns an array of the same type as the stack. That is, if a stack is configured to hold strings, the array will be an array of strings.

The stack has a `Peek` method that allows the retrieval of the top element from the stack without it being removed:

```
$stack.Peek()
```

The `Peek` method throws an error if the stack is empty (refer to the *Removing elements from the stack* section).

Adding elements to the stack

Elements may be added to the stack using the `Push` method:

```
$stack.Push("Up the road")
$stack.Push("Over the gate")
$stack.Push("Under the bridge")
```

Removing elements from the stack

Elements may be removed from the stack using the `Pop` method:

```
$stack.Pop()    # This returns Under the bridge
```

If the stack is empty and the `Pop` method is called, an error is thrown, as shown here:

```
PS> $stack.Pop()
MethodInvocationException: Exception calling "Pop" with "0" argument(s): "Stack empty."
```

To avoid this, the `Count` property of the stack may be inspected, for example:

```
# Set-up the stack
$stack = [System.Collections.Generic.Stack[String]]::new()
"Up the road", "Over the gate", "Under the bridge" | ForEach-Object {
    $stack.Push($_)
}
# Pop from the stack until the stack is empty
while ($stack.Count -gt 0) {
    Write-Host $stack.Pop()
}
```

Summary

In this chapter, we learned that variables can be created to contain information that is to be used in a function or a script.

Variable commands are available to interact with variables beyond changing the value, such as setting a description, making a variable in a specific scope, or making a variable private.

A variable scope affects how variables may be accessed. Variables are created in the local scope by default. Arrays are sets of objects of the same type. Arrays are immutable, and the size of an array cannot change after creation. Adding or removing elements from an array requires the creation of a new array. Hashtables are associative arrays. An element in a Hashtable is accessed using a unique key.

Lists, stacks, queues, and dictionaries are advanced collections that may be used when a particular behavior is required, or if they offer a desirable performance benefit. *Chapter 6, Conditional Statements and Loops*, explores branching and looping in PowerShell.

6

Conditional Statements and Loops

Conditional statements and loops are the backbone of any programming or scripting language. The ability to react to a state and choose a path to take or the ability to repeat instructions is vital.

Each conditional statement or loop creates a branch in a script or piece of code. The branch represents a split in the instruction set. Branches can be conditional, such as one created by an `if` statement, or unconditional, such as a `foreach` loop. As the number of branches increases, so does the complexity. The paths through the script spread out in the same manner as the limbs of a tree.

Statements or lines of code may be executed when certain conditions are met. PowerShell provides `if` and `switch` statements for this purpose. Loops allow code to be repeated for a set of elements or until a specific condition is met.

In this chapter, the following topics are covered:

- `if`, `else`, and `elseif`
- `switch`
- Loops
- `break` and `continue`
- Implicit Boolean

Let's look at `if` statements first.

if, else, and elseif

An `if` statement is used to execute an action when a condition is met. The following shows the syntax for an `if` statement; the statements enclosed by the `if` statement executes if the condition evaluates to `true`:

```
if (<condition>) {
    <statements>
}
```

The `else` statement is optional and runs if all previous conditions evaluate to `false`:

```
if (<first-condition>) {
    <first-statements>
} else {
    <second-statements>
}
```

The `elseif` statement allows several conditions to be tested in order:

```
if (<first-condition>) {
    <first-statements>
} elseif (<second-condition>) {
    <second-statements>
} elseif (<last-condition>) {
    <last-statements>
}
```

The `else` statement may be added after any number of `elseif` statements.

Execution of a block of conditions stops as soon as a single condition evaluates to `true`. For example, both the first and second condition would evaluate to `true`, but only the first executes:

```
$value = 1
if ($value -eq 1) {
    Write-Host 'value is 1'
} elseif ($value -lt 10) {
    Write-Host 'value is less than 10'
}
```

If the `if` statement structure becomes too complicated, a `switch` statement might be used instead. Switch is explored later in this chapter.

Assignment within if statements

Values may be assigned to variables inside `if` statements as shown here:

```
if ($i = 1) {
    Write-Host "The variable i is $i"
}
```

The condition will be `true` if the value of the variable evaluates to `true`.

This is most used when testing for the existence of a value in a variable, for example:

```
if ($interface = Get-NetAdapter | Where-Object Status -eq 'Up') {
    Write-Host "$($interface.Name) is up"
}
```

In the previous example, the statement to the right of the assignment operator (`=`) is executed, assigned to the `$interface` variable, and then the value in the variable is treated as an implicit Boolean.

Supporting this type of assignment means PowerShell does not alert the author of a script if an assignment operation is used instead of a comparison. This is most often a problem when the assignment was intended instead to be a comparison.

switch

A `switch` statement executes statements where a case evaluates to `true`. The case can be a number, a string, or any other value. Switch is similar in some respects to an `if`, `elseif`, or `else` statement. The key difference is that, in `switch`, more than one condition can match the value being tested.

A `switch` statement uses the following generalized notation:

```
switch [-regex|-wildcard|-exact][-casesensitive] (<value>) {
    <case>   { <statements> }
    <case>   { <statements> }
    default  { <statements> }
}
```

The value is the "subject" of the `switch` statement; it is compared to each of the cases in turn.

The `casesensitive` parameter applies when the cases are strings.

Each case is evaluated in turn and, by default, any matching cases will execute. The `default` case is optional and will only be executed if no other case matches as shown here:

```
$value = 2
switch ($value) {
    1       { Write-Host 'value is 1' }
    default { Write-Host 'No conditions matched' }
}
```

Within the `switch` statement, the variable `$_` or `$PSItem` may be used to refer to the value. If `switch` is enclosed in `ForEach-Object`, for example, the value of `$_` may differ from the value `ForEach-Object` holds in the process block.

switch and arrays

The `switch` statement can be used on both scalar (single) values and arrays of values. If the value is an array, then each case will be tested against each element of the array:

```
$arrayOfValues = 1..3
switch ($arrayOfValues) {
    1 { 'One' }
    2 { 'Two' }
    3 { 'Three' }
}
```

As `switch` is technically a loop, it does not execute at all if given an empty array (such as `@()`). In the following example, an empty array is the value. As the value is empty, and as `switch` is a loop, none of the cases run (including `default`):

```
switch (@()) {
    default { 'this default case will not run' }
}
```

`switch`, like other loops, executes if an explicit `$null` value is used. For example:

```
switch ($null) {
    default { 'this default case will run' }
}
```

The ability to act on an array allows `switch` to act on file content.

switch and files

The `switch` statement can also be used to work on the content of a file using the following notation:

```
switch [-regex|-wildcard][-casesensitive] -File <Name> {
    <condition> { <statements> }
    <condition> { <statements> }
}
```

The `File` parameter can be used to select from a text file (line by line); the content of the file is read as an array.

wildcard and regex

The `wildcard` and `regex` parameters are used when matching strings.

The `wildcard` parameter allows the use of the characters `?` (any single character) and `*` (any string of characters) in a condition, for example:

```
switch -Wildcard ('cat') {
    'c*'  { Write-Host 'The word begins with c' }
    '???' { Write-Host 'The word is 3 characters long' }
    '*t'  { Write-Host 'The word ends with t' }
}
```

The `Regex` parameter allows for the use of regular expressions to perform comparisons (*Chapter 9, Regular Expressions*, explains this syntax in greater detail), for example:

```
switch -Regex ('cat') {
 '^c'          { Write-Host 'The word begins with c' }
  '^[a-z]{3}$' { Write-Host 'The word is 3 characters long' }
  't$'         { Write-Host 'The word ends with t' }
}
```

The `switch` statement also allows the cases to be defined as a script block.

Script block cases

`switch` allows a script block to be used in place of the simpler direct comparison cases used in the preceding sections. The script block is executed, and the result determines if the case is matched. For example:

```
switch (Get-Date) {
    { $_ -is [DateTime] } { Write-Host 'This is a DateTime type' }
    { $_.Year -ge 2020 }  { Write-Host 'It is 2020 or later' }
}
```

Script block cases like those used above may be mixed with other values. The following example uses a single `ScriptBlock` expression as well as a comparison with the value `5`:

```
$Value = 5
switch ($Value) {
    { $_ -is [Int] } { Write-Host 'This is a Int32 type' }
    5                { Write-Host 'The value is five' }
}
```

Expressions like the preceding example might replace a more complex `if-elseif` structure or they may be used to perform more complex comparison combinations.

For example, shared code might be established to use with more than one case:

```
function Set-FileState {
    param (
        [Parameter(Mandatory)]
        [ValidateSet('Update', 'Create', 'Delete')]
        [string]$Action
    )
    $params = @{
        Path = '~\test.txt'
    }
    switch ($Action) {
        { $_ -in 'Update', 'Create' } {
            $commonParams['Value'] = 'File content'
        }
        'Update' { Set-Content @params }
        'Create' { New-Item @params -ItemType File }
        'Delete' { Remove-Item @params }
    }
}
Set-FileState -Action Update
```

Inside a `switch` statement, `$_` is the value that is being tested. It is possible to re-assign the value of `$_` in one case, allowing a second case (or any other cases) to be applied:

```
function Set-FileState {
    param (
        [Parameter(Mandatory)]
        [ValidateSet('Update', 'Create', 'Delete')]
        [string]$Action
    )
    $params = @{
        Path = '~\test.txt'
        ItemType = 'File'
        Value    = 'File content'
    }
    switch ($Action) {
        { $_ -in 'Delete', 'Update' } {
            if (Test-Path -Path $params.Path) {
                Remove-Item -Path $params.Path
            }
            if ($_ -eq 'Update') {
                $_ = 'Create'
            }
        }
        'Create' {
            New-Item @params
        }
    }
}
Set-FileState -Action Update
```

The `switch` statement will run every case that matches the value in turn until the last is reached. The `break` and `continue` keywords may be used to affect how a `switch` statement ends.

Switch, break, and continue

By default, `switch` executes every case where the case evaluates as `true` when compared with the value.

The `break` and `continue` keywords may be used within the `switch` statement to control when it should stop testing a value. `break` ends the `switch` statement; `continue`, when the value is an array, continues to the next element in the array.

`break` is often most appropriate if the value is a scalar, and `continue` when the value is an array. However, either may be used as needed.

The `switch` statement will not stop testing conditions unless the `break` keyword is used, for example:

```
$value = 1
switch ($value) {
    1 { Write-Host 'value is 1' }
    1 { Write-Host 'value is still 1' }
}
```

In the following example, where `switch` is acting on an array, both statements will execute:

```
switch (1, 2) {
    1 { Write-Host 'Equals 1' }
    2 { Write-Host 'Equals 2' }
}
```

If the `break` keyword is included, as shown here, only the first executes then the `switch` statement stops:

```
switch (1, 2) {
    1 { Write-Host 'Equals 1'; break }
    2 { Write-Host 'Equals 2' }
}
```

The first example in this section has two cases that match the value `1`. If `continue` is used, the second matching statement is skipped, and `switch` continues to the next element in the array.

```
switch (1, 2) {
    1 { Write-Host 'Equals 1'; continue }
    1 { Write-Host 'value is still 1' }
    2 { Write-Host 'Equals 2' }
}
```

Finally, `break` and `continue` can be mixed if necessary. In the following example, the condition for the value `3` will never be reached if the array contains the value `2`.

```
switch (1, 2, 3) {
    1 { Write-Host 'One'; continue }
    1 { Write-Host 'One again' }
    2 { Write-Host 'Two'; break }
    3 { Write-Host 'Three' }
}
```

The `switch` statement is incredibly flexible and allows relatively complex condition structures to be described in a concise manner.

PowerShell has several other loop keywords that may be used to repeat statements.

Loops

Loops may be used to iterate through collections, performing an operation against each element in the collection, or to repeat an operation (or series of operations) until a condition is met.

The following loops will be demonstrated in this section:

- `foreach`
- `for`
- `do`
- `while`

The `foreach` loop is perhaps the most common of these loops.

Foreach loop

The `foreach` loop executes against each element of a collection using the following notation:

```
foreach (<element> in <collection>) {
    <statements>
}
```

For example, the `foreach` loop may be used to iterate through each of the processes returned by `Get-Process`:

```
foreach ($process in Get-Process) {
    Write-Host $process.Name
}
```

If the collection is `$null` or empty, the body of the loop will not execute.

> **foreach and foreach**
>
> PowerShell comes with the alias `foreach` for the `ForEach-Object` command. When this alias acts depends on context.
>
> If `foreach` is first in a statement, then the loop keyword is used.
>
> ```
> foreach ($value in $array) { }
> ```
>
> If `foreach` is placed after a pipe then the alias is used, the `ForEach-Object` (`foreach`) is used.
>
> `$array | foreach { }`
>
> Because this can cause confusion, it is rarely a good idea to use the `foreach` alias.

for loop

The `for` loop is typically used to step through a collection using the following notation:

```
for (<initial>; <condition>; <repeat>){
    <body-statements>
}
```

`<initial>` runs before the loop starts and is normally used to set the initial state, normally the variables used within the loop.

The loop continues to run as long as `<condition>` evaluates to `true`.

The `<repeat>` block is executed after each iteration of the loop.

The following example shows the most common use of the `for` loop:

```
$processes = Get-Process
for ($i = 0; $i -lt $processes.Count; $i++) {
    Write-Host $processes[$i].Name
}
```

The `for` loop provides a significant degree of control over the loop and is useful where the increment or the actions inside the loop need to be completed in an order other than a simple ascending order. The examples that follow show some of the possible ways in which the `for` loop can be used.

For example, you can use the `<repeat>` criteria to execute the body for every third element:

```
for ($i = 0; $i -lt $processes.Count; $i += 3) {
    Write-Host $processes[$i].Name
}
```

Or you can set the `<initial>` parameters to start at the end and work toward the beginning of an array, for example:

```
for ($i = $processes.Count - 1; $i -ge 0; $i--) {
    Write-Host $processes[$i].Name
}
```

As the body of the loop in the preceding example has access to the index, it can easily access elements adjacent to the current element. The following example reads two characters at a time from `$encodedString`:

```
$encodedString = '68656C6C6F20776F726C64'
[char[]]$characters = for ($i = 0; $i -lt $encodedString.Length; $i += 2) {
    $hex = '0x{0}{1}' -f @(
        $encodedString[$i]
        $encodedString[$i + 1]
    )
    1 * $hex
}
[string]::new($characters)
```

Running the preceding example will allow PowerShell to convert each pair of characters into a hexadecimal number based on `$i` incrementing two at a time. The result of the preceding loop is a single string.

Each element in the `for` loop is optional; the following loop only ends because the body contains `break`:

```
for (;;) {
    break
}
```

Any of the three elements, initial, condition, and repeat may be included individually or in any combination to describe how the loop should act.

The `for` loop can also act on more than one variable at a time although it is incredibly rare to find this used in practice. In the following example, two variables are given an initial state, and two variables are incremented after each iteration of the loop:

```
for (($i = 0), ($j = 0); $i -le 10 ; $i++, ($j += 2)) {
    Write-Host "$i :: $j"
}
```

The preceding condition only tests the state of one variable in the preceding example, but a more complex expression might have been used, for example:

```
$i -le 10 -and $j -le 20
```

The preceding example works because the different elements of the `for` loop, initial, condition, and repeat are arbitrary blocks of code.

do until and do while loops

`do until` and `do while` each execute the body of the loop at least once, as the condition test is at the end of the loop statement. Loops based on `do until` will exit when the condition evaluates to `true`; loops based on `do while` will exit when the condition evaluates to `false`.

`do` loops are written using the following notation:

```
do {
    <body-statements>
} <until | while> (<condition>)
```

`do until` is suited to exit conditions that are expected to be positive. Using `until` avoids the need to test for a false value in the condition. For example, a script might wait for a computer to respond to a ping:

```
do {
    Write-Host "Waiting for boot"
    Start-Sleep -Seconds 5
} until (Test-Connection 'SomeComputer' -Quiet -Count 1)
```

The previous example can be written using `do while`, but the condition is slightly more complex and may be harder to read:

```
do {
    Write-Host "Waiting for boot"
    Start-Sleep -Seconds 5
} while (-not (Test-Connection 'SomeComputer' -Quiet -Count 1))
```

The `do while` loop is more suitable for exit conditions that are negative. For example, a loop might wait for a remote computer to stop responding to a ping:

```
do {
    Write-Host "Waiting for shutdown"
    Start-Sleep -Seconds 5
} while (Test-Connection 'SomeComputer' -Quiet -Count 1)
```

While loop

As the condition for a `while` loop comes first, the body of the loop will only execute if the condition evaluates to `true`:

```
while (<condition>) {
    <body-statements>
}
```

A `while` loop may be used to wait for something to happen. In the following example, the loop continues until a file exists:

```
while (-not (Test-Path $env:TEMP\test.txt -PathType Leaf)) {
    Start-Sleep -Seconds 10
}
```

Loops, break, and continue

The `break` keyword can be used to end a loop early. The loop in the following example would continue until the value of `$i` is `20`. `break` is used to stop the loop when `$i` reaches `10`:

```
for ($i = 0; $i -lt 20; $i += 2) {
    Write-Host $i
    if ($i -eq 10) {
        break    # Stop this loop
    }
}
```

The `break` keyword acts on the loop it is nested inside. In the following example, the `do` loop breaks early—when the variable `$i` is less than or equal to `2` and variable `$k` is greater than or equal to `3`:

```
$i = 1 # Initial state for i
do {
    Write-Host "i: $i"
    $k = 1 # Reset k
    while ($k -lt 5) {
        Write-Host "  k: $k"
        $k++ # Increment k
        if ($i -le 2 -and $k -ge 3) {
            break
        }
    }
    $i++ # Increment i
} while ($i -le 3)
```

The output of the loop is:

```
i: 1
  k: 1
  k: 2
i: 2
  k: 1
  k: 2
i: 3
  k: 1
  k: 2
  k: 3
  k: 4
```

The `continue` keyword may be used to move on to the next iteration of a loop immediately. For example, the following loop executes a subset of the loop body when the value of the variable `$i` variable is less than `2`:

```
for ($i = 0; $i -le 5; $i++) {
    Write-Host $i
    if ($i -lt 2) {
        continue    # Continue to the next iteration
    }
    Write-Host "Remainder when $i is divided by 2 is $($i % 2)"
}
```

Break and continue outside loops

The `break` and `continue` keywords should not be used except within a loop or a `switch` statement. If `break` is used outside a loop PowerShell will look through any parent scopes (to Global scope) until it finds a loop to stop or runs out of scopes to search.

In the following example, `break` is erroneously used to end a function early.

```
function Test-Value {
    [CmdletBinding()]
    param (
        [int]$Value
    )
    if ($Value -eq 7) {
        break
    }
    $true
}
```

A casual test may suggest that `break` is only stopping the function, and everything is working as it should. However, if the function is used within a loop, `break` will affect that loop. This happens no matter how many scopes (or other functions) there are between the scope below and the function's scope.

```
foreach ($value in 1..10) {
    Write-Verbose "Working on $value" -Verbose
    if ($value -gt 5) {
        if (Test-Value $value) {
            Write-Host "$value is OK"
        }
    }
}
```

The script should repeat the `Write-Host` statement for 6, 8, 9, and 10. However, because `break` is used in the function, the `foreach` loop stops as soon as it reaches 7.

The `continue` keyword has the same problem although this will continue to the next iteration of the loop in the parent scope instead of terminating the loop.

Return and exit

The `return` and `exit` keywords are also the subject of misuse seen with `break`.

The `return` keyword can be used to end a `ScriptBlock` early. For example, it might be used in place of `continue` inside the `Process` argument of `ForEach-Object` as shown here.

```
1..10 | ForEach-Object {
    if ($_ -lt 5) { return }
    $_
}
```

The `exit` keyword, when used in a script, will end the script immediately. But when `exit` is used in a module or function, it will close the PowerShell session. The `return` keyword is a safer choice when the intent is to end a script or function early.

If `break` and `continue` are used inside a nested loop (a loop within a loop), a specific loop can be targeted by creating and using a label for a loop.

Loops and labels

In PowerShell, a loop can be given a label. The label may be used with `break` and continue to define a specific loop to `break` from or `continue` to.

The label is written before the loop keyword (`for` , `while` , `do` , or `foreach`) and is preceded by a colon character. For example:

```
:ThisIsALabel foreach ($value in 1..10) {
    $value
}
```

The label may be placed directly before the loop keyword or on the line above. White space may appear between the label and the loop keyword.

The label is used with either `break` or `continue` and can be useful when one loop is nested inside another. The label name is written after the `break` or `continue` keyword, and the colon should be excluded.

```
:outerLoop for ($i = 1; $i -le 5; $i++) {
    :innerLoop foreach ($value in 1..5) {
        Write-Host "$i :: $value"
        if ($value -eq $i) {
            continue outerLoop
        }
    }
}
```

The `break` may be used in a similar manner to end a labeled loop.

Implicit Boolean

A value that is not Boolean but is tested as if it were `true` or `false` is an implicit Boolean. For example, the following `if` statement tests the output from a command:

```
if (Get-ChildItem c:\users\a*) {
    # If statement body
}
```

The condition evaluates as `true` when the `Get-ChildItem` command finds one or more files or folders. The condition evaluates as `false` when no files or folders are found.

An explicit version of the same comparison is shown here:

```
if ($null -ne (Get-ChildItem c:\users\c*)) {
    # If statement body
}
```

The previous explicit statement is clearly more complex and more difficult to read.

A condition with no comparison operator implicitly evaluates to `false` if it is any of the following:

- `$null`
- An empty string
- An empty array
- The numeric value 0

A variable containing a single object, or an array containing one or more elements, and so on evaluate to `true`.

Summary

This chapter explored the different conditional and looping statements available in PowerShell.

The `if` statement allows statements to be run when a condition is met and may be extended to test several conditions with `elseif`.

The `switch` statement has similarities with `if`, a comparison of one value against another. However, `switch` can test many individual cases expressed in a concise manner, allows more than one case to apply to a value, and can operate on an array.

Looping is a vital part of any programming language and PowerShell is no exception. The `foreach` loop is perhaps the most used, allowing repeated code to be enclosed and executed against several objects. The `foreach` loop keyword can often be replaced with the `ForEach-Object` command (or vice versa) depending on circumstances, but the two should not be confused.

The `for` loops are more complex but offer a great deal of flexibility for any operation based on a numeric sequence. The `while` and `do` loops can be used to carry on working until a condition is met, for example, waiting for a timeout, or an item is created.

7

Working with .NET

Microsoft .NET is an extensive library of **APIs** (**Application Program Interfaces**, or pre-created code) that can be used by developers when writing applications, or scripters when writing scripts. Microsoft has created several different versions of .NET over the years starting with .NET Framework 1.0 released in 2002. .NET Framework is limited to the Windows operating system.

Windows PowerShell is built using .NET Framework. The .NET Framework version depends on the local installation and the PowerShell version. PowerShell 5.1 was built using .NET 4.5 and can make use of .NET 4.8 if it is installed: `https://docs.microsoft.com/dotnet/api/?view=netframework-4.8`.

In 2016, Microsoft released .NET Core, a release of .NET that could be run across different platforms, including Windows, Linux distributions, and macOS.

PowerShell 6 and PowerShell 7.0 were built using .NET Core, that is, .NET Core 3.1 for PowerShell 7.0. PowerShell 7.0 can therefore make use of many of the APIs in .NET Core 3.1: `https://docs.microsoft.com/dotnet/api/?view=netcore-3.1`.

In November 2020, Microsoft released .NET 5. .NET 5 extends on .NET Core. "Core" was removed from the name to indicate a merging of the two distinct releases of .NET. Going forward there is no more .NET Framework (Windows only) and .NET Core (cross-platform); there is just .NET, and that is cross-platform.

PowerShell 7.1 was built using .NET 5 and can make use of the APIs: `https://docs.microsoft.com/dotnet/api/?view=net-5.0`.

The ability to consume .NET APIs adds a tremendous amount of flexibility to PowerShell over and above the commands provided by PowerShell itself or any modules that can be installed.

The concept of working with objects was introduced in *Chapter 3*, *Working with Objects in PowerShell*. This chapter extends on working with objects, moving from objects created by commands to objects created from .NET classes.

Working directly with .NET is important because as a PowerShell developer you do not want to be limited to the things that have been turned into neat little commands. Being able to directly use these types opens the possibility of using third-party assemblies, such as those on `nuget.org`: `https://www.nuget.org/packages`.

It is important to understand that .NET is vast; it is not possible to cover everything about .NET in a single chapter. This chapter aims to show how .NET may be used within PowerShell based on the Microsoft Docs references: `https://docs.microsoft.com/dotnet/api/?view=netcore-3.1`.

The goal is therefore not to learn everything about .NET, but to learn enough know-how to learn more.

This chapter covers the following topics:

- Assemblies
- Types
- Enumerations
- Classes
- Namespaces
- The `using` keyword
- Type accelerators
- Members
- Fluent interfaces
- Reflection in PowerShell

Assemblies are the starting point; they contain the types that can be used in PowerShell.

Assemblies

An assembly is a collection of types and any other supporting resources. .NET objects are implemented within assemblies. An assembly may be static (based on a file) or dynamic (created in memory).

The assembly type load locations can be seen by exploring the `Assembly` property of the type. For example, the `String` type is loaded from `System.Private.CoreLib.dll` in PowerShell 7:

```
PS> [System.String].Assembly.Location
C:\Program Files\PowerShell\7\System.Private.CoreLib.dll
```

In PowerShell 7, the assemblies that are loaded by default or those that can be loaded by name are in the `$PSHome` directory.

You can view the list of currently loaded assemblies in a PowerShell session using the following statement:

```
[System.AppDomain]::CurrentDomain.GetAssemblies()
```

The list can be quite extensive and can grow as different modules (which might depend on other .NET types) are loaded. The first few lines are shown here:

```
GAC    Version     Location
---    -------     --------
False  v4.0.30319  C:\Program Files\PowerShell\7\System.Private.CoreLib.dll
False  v4.0.30319  C:\Program Files\PowerShell\7\pwsh.dll
False  v4.0.30319  C:\Program Files\PowerShell\7\System.Runtime.dll
False  v4.0.30319  C:\Program Files\PowerShell\7\Microsoft.PowerShell.Co...
False  v4.0.30319  C:\Program Files\PowerShell\7\System.Management.Autom...
False  v4.0.30319  C:\Program Files\PowerShell\7\System.Threading.Thread...
False  v4.0.30319  C:\Program Files\PowerShell\7\System.Runtime.InteropS...
False  v4.0.30319  C:\Program Files\PowerShell\7\System.Threading.dll
```

You can use the `ClassExplorer` module from the PowerShell Gallery to simplify the previous command:

```
Install-Module ClassExplorer
Get-Assembly
```

Assemblies can be explicitly loaded with the `Add-Type` command. PowerShell 7 includes the `System.Windows.Forms.dll` file in the `$PSHome` folder but will only load it if told to. The DLL is used to write some graphical user interfaces. You can therefore load it using the name alone (instead of a full path to the DLL):

```
Add-Type -AssemblyName System.Windows.Forms
```

Once an assembly, and the types it contains, have been loaded into a session, they cannot be unloaded without completely restarting the PowerShell session. This might affect an upgrade to an existing module based on a DLL; PowerShell cannot unload the DLL and load a newer version. When a DLL is in use by an application, it is locked; attempting to delete the DLL file would fail.

Much of PowerShell itself is implemented in the `System.Management.Automation` DLL. You can view details of the DLL using the following statement:

```
[System.Management.Automation.PowerShell].Assembly
```

In this statement, the `PowerShell` type is used to get information about the `System.Management.Automation` assembly. The `PowerShell` type will be used in *Chapter 15, Asynchronous Processing*.

Any other type in the same assembly may be used to get the same `Assembly` property. The `PowerShell` type could be replaced with any other type implemented as part of PowerShell itself:

```
[System.Management.Automation.PSCredential].Assembly
[System.Management.Automation.PSObject].Assembly
```

In Windows PowerShell, assemblies are often loaded from the **Global Assembly Cache** (**GAC**). PowerShell 7 (and other .NET Core applications) cannot use the GAC, which is why the GAC property in each of the assemblies in use by PowerShell 7 is `False`.

About the Global Assembly Cache

In Windows PowerShell, most types exist in DLL files stored in `%SystemRoot%\Assembly`. This folder stores what is known as the **Global Assembly Cache** (**GAC**). The DLL files registered here can be used by any .NET Framework application on the computer. Each DLL may be used by name, rather than an application needing to know the exact path to a DLL.

You can use the `Gac` module, in the PowerShell Gallery, to list assemblies. The versions of an assembly in the GAC will vary depending on the installed versions of .NET Framework (and any other installed components, such as **Software Development Kits** or **SDKs**):

```
PS> Install-Module Gac -Scope CurrentUser
PS> Get-GacAssembly System.Windows.Forms
Name                 Version    Culture PublicKeyToken   PrArch
----                 -------    ------- --------------   ------
System.Windows.Forms 2.0.0.0            b77a5c561934e089 MSIL
System.Windows.Forms 1.0.5000.0         b77a5c561934e089 None
System.Windows.Forms 4.0.0.0            b77a5c561934e089 MSIL
```

If a specific version is required, you can use the full name of the assembly as it uniquely identifies the assembly:

```
Add-Type -AssemblyName 'System.Windows.Forms, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089'
```

You can use the `Gac` module to show the `FullName` value used in the previous code. The values displayed depend on the installed versions of .NET Framework:

```
PS> Get-GacAssembly System.Windows.Forms | Select-Object FullName
FullName
--------
System.Windows.Forms, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
System.Windows.Forms, Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
System.Windows.Forms, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

An assembly typically contains several different types.

Types

A type is used to represent the generalized functionality of an object. This description is vague, but a .NET type can be used to describe anything; it is hard to be more specific. To use this book as an example, it could have several types, including the following:

- `PowerShellBook`
- `TextBook`
- `Book`

Each of these types describes how an object can behave. The type does not describe how this book came to be, or whether it will

do anything (on its own) to help create one.

In PowerShell, types are written between square brackets. The `[System.AppDomain]` and `[System.Management.Automation.PowerShell]` statements, used when discussing previous assemblies, are types.

The type of an object can be revealed by the `Get-Member` command (the following output is truncated):

```
PS> 1 | Get-Member
   TypeName: System.Int32
Name        MemberType Definition
----        ---------- ----------
CompareTo   Method     int CompareTo(System.Object value), int ...
Equals      Method     bool Equals(System.Object obj), bool Equ...
GetHashCode Method     int GetHashCode()
```

You can also use the `GetType` method, for example, by using the method on a variable:

```
PS> $variable = 1
PS> $variable.GetType()
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Int32                                    System.ValueType
```

Methods are explored later in this chapter.

Type descriptions are objects in PowerShell

`[System.AppDomain]` is a type, but the syntax used to describe the type is itself an object. The object has properties and methods and a type of its own (`RuntimeType`).

This can be seen by running the following command:

```
[System.AppDomain].GetType()
```

You can create types using several different keywords, including `enum`, `struct`, `interface`, and `class`. `struct` and `interface` types are beyond the scope of this chapter.

Enumerations

An enumeration is a specialized type that is used to express a list of constants. Enumerations are used throughout .NET and PowerShell.

PowerShell itself makes use of enumerations for many purposes. For example, the possible values for the `$VerbosePreference` variable are described in an enumeration:

```
PS> $VerbosePreference.GetType()
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     ActionPreference                         System.Enum
```

Notice that the `BaseType` is `System.Enum`, indicating that this type is an enumeration.

The possible values for an enumeration can be listed in several different ways. The most convenient of these is to use the `GetEnumValues()` method on the enumeration type:

```
PS> $VerbosePreference.GetType().GetEnumValues()
SilentlyContinue
Stop
Continue
Inquire
Ignore
Suspend
Break
```

Enumerations are relatively simple types. They contain a list of constants that you can use in your code. A class is more complex.

Classes

A class is a set of instructions that dictates how a specific instance of an object behaves, including how it can be created and what it can do. A class is, in a sense, a recipe.

In the case of this book, a class might include details of authoring, editorial processes, and publication steps. These steps are, hopefully, invisible to anyone reading this book; they are part of the internal implementation of the class. Following these steps will produce an instance of the `PowerShellBook` object.

Once a class has been compiled, it is used as a type. The members of the class are used to interact with the object. Members are explored later in this chapter.

Classes (and types) are arranged and categorized into namespaces.

Namespaces

A namespace is used to organize types into a hierarchy, grouping types with related functionality together. A namespace can be considered like a folder in a file system.

PowerShell is for the most part implemented in the `System.Management.Automation` namespace. This namespace has associated documentation: https://docs.microsoft.com/dotnet/api/system.management.automation?view=powershellsdk-7.0.0.

Similarly, types used to work with the filesystem are grouped together in the `System.IO` namespace: https://docs.microsoft.com/dotnet/api/system.io.

For the following given type name, the namespace is everything before the final label. The namespace value is accessible as a property of the type:

```
PS> [System.IO.File].Namespace
System.IO
```

In PowerShell, the `System` namespace is implicit. The `System.AppDomain` type was used at the start of the chapter to show which assemblies PowerShell is currently using. This can be shortened to:

```
[AppDomain]::CurrentDomain.GetAssemblies()
```

The same applies to types with longer names, such as `System.Management.Automation.PowerShell`, which can be shortened to:

```
[Management.Automation.PowerShell].Assembly
```

PowerShell automatically searches the `System` namespace for these types. The `using` keyword can be used to look up types in longer namespaces.

The using keyword

The `using` keyword simplifies the use of namespaces and can be used to load assemblies or PowerShell modules. The `using` keyword was introduced with PowerShell 5.0.

You can use the `using` keyword in a script, a module, or the console. In a script, the `using` keyword can only be preceded by comments.

The `using module` statement is used to access PowerShell classes created within a PowerShell module. The `using module` statement is explored in *Chapter 19, Classes and Enumerations*.

In the context of working with .NET, namespaces and assemblies are of interest.

Using namespaces

The `using namespace` statement instructs PowerShell to look for any type names used in an additional namespace. For example, by default, attempting to use `System.IO.File` without a full name will result in an error:

```
PS> [File]
InvalidOperation: Unable to find type [File].
```

PowerShell looked for the type in the `System` namespace and did not find it.

If `using namespace System.IO` is added, first PowerShell will search the `System.IO` namespace for the type (in addition to the top-level namespace `System`):

```
PS> using namespace System.IO
PS> [File]
IsPublic IsSerial Name                     BaseType
-------- -------- ----                     --------
True     False    File                     System.Object
```

In the console, PowerShell only recognizes the last `using namespace` statement that was entered. In the following example, the first `using namespace` statement is replaced by the second when typed into the console.

```
using namespace System.IO
using namespace System.Data.SqlClient
```

PowerShell will be able to find the type `System.Data.SqlClient.SqlConnection` by name only as the following code shows:

```
PS> [SqlConnection]
IsPublic IsSerial Name                     BaseType
-------- -------- ----                     --------
True     False    SqlConnection            System.Data.Common.DbConnection
```

But PowerShell will fail to find the `[File]` type again because the previous `using namespace` statement is no longer valid:

```
PS> [File]
InvalidOperation: Unable to find type [File].
```

A script file allows multiple `using namespace` statements, which are processed when the script is parsed. In the console, it is only possible to have more than one `using namespace` statement if they are separated by a semi-colon:

```
using namespace System.IO; using namespace System.Data.SqlClient
```

After which, PowerShell will search `System.Data.SqlClient`, `System.IO`, and `System` when a type is entered by name only.

The namespace value used with a `using namespace` statement does not have to exist and PowerShell does not attempt to validate the value. This means that an assembly that implements the types in a namespace can be loaded after the `using namespace` statement.

For example, if a new PowerShell session is started and the `System.Windows.Forms` DLL has not yet loaded, the following commands can be executed and PowerShell will find types by name from the loaded assembly:

```
PS> using namespace System.Windows.Forms
PS> Add-Type -AssemblyName System.Windows.Forms
PS> [Button]
IsPublic IsSerial Name                      BaseType
-------- -------- ----                      --------
True     False    Button                    System.Windows.Forms.ButtonBase
```

It is possible to load assemblies (before or after `using namespace`) with the `using assembly` statement.

Using assemblies

The `using assembly` statement is used to load assemblies into the PowerShell session.

In PowerShell 7, `using` `assembly` can only load assemblies using a path (full or relative). In Windows PowerShell, assemblies can either be loaded using a path or from the GAC.

For example, you can load the `System.Windows.Forms` assembly in Windows PowerShell with the following command:

```
using assembly System.Windows.Forms
```

In PowerShell 7, you have to use the full path. The path would have to be written in full as variables are not permitted in the statement:

```
using assembly 'C:\Program Files\PowerShell\7\System.Windows.Forms.dll'
```

PowerShell 7 may only be able to load by name in the future, but a GitHub issue about this has been open for 2 years.

PowerShell allows you to run the `using assembly` statement any number of times in a script, and more than one assembly can be loaded in a single script.

Before `using namespace` became available with PowerShell 5, the only way to shorten a type name was to use a type accelerator.

Type accelerators

A type accelerator is an alias for a type. At the beginning of this chapter, the `System.Management.Automation.PowerShell` type was used; this type has an accelerator available. The accelerator name is simply `PowerShell`. The accelerator allows the following to be used:

```
[PowerShell].Assembly
```

Another commonly used example is the `ADSI` accelerator. This represents the `System.DirectoryServices.DirectoryEntry` type. This means that the following two commands are equivalent:

```
[System.DirectoryServices.DirectoryEntry]"WinNT://$env:COMPUTERNAME"
[ADSI]"WinNT://$env:COMPUTERNAME"
```

The full type name behind a type accelerator can be seen using the `FullName` property:

```
PS> [ADSI].FullName
System.DirectoryServices.DirectoryEntry
```

PowerShell includes a lot of type accelerators; these accelerators are documented in `about_Type_Accelerators`:

```
Get-Help about_Type_Accelerators
```

Types have members; the members available depend on the type. Types derived from classes are likely to have several different members, including constructors, properties, and methods.

Members

All types have members. Members represent data, or dictate the behavior of the object represented by a type.

In .NET, the members are described in the C Sharp (C#) programming guide on Microsoft Docs: https://docs.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/members.

PowerShell also adds members; these member types are also described on Microsoft Docs: https://docs.microsoft.com/dotnet/api/system.management.automation.psmembertypes.

This section focuses on a small number of members used when working with .NET types:

- Constructors
- Methods
- Properties

The `Event` member is explored in *Chapter 15, Asynchronous Processing*.

The `ScriptProperty` and `ScriptMethod` property types are specific to PowerShell and may be added with the `Add-Member` command. These member types are outside of the scope of this chapter.

Constructors are one possible way of creating an instance of a type.

Constructors

A constructor is used to create an instance of a type. For example, the `System.Text.StringBuilder` type can be used to build complex strings.

The `StringBuilder` class is documented on Microsoft Docs: https://docs.microsoft.com/dotnet/api/system.text.stringbuilder.

The **StringBuilder Constructors** section of the `StringBuilder` class documentation has several constructors as shown in *Figure 7.1*:

StringBuilder Constructors

Namespace: System.Text

Assembly: System.Runtime.dll

Initializes a new instance of the StringBuilder class.

Overloads

StringBuilder()	Initializes a new instance of the StringBuilder class.
StringBuilder(Int32)	Initializes a new instance of the StringBuilder class using the specified capacity.
StringBuilder(String)	Initializes a new instance of the StringBuilder class using the specified string.
StringBuilder(Int32, Int32)	Initializes a new instance of the StringBuilder class that starts with a specified capacity and can grow to a specified maximum.
StringBuilder(String, Int32)	Initializes a new instance of the StringBuilder class using the specified string and capacity.
StringBuilder(String, Int32, Int32, Int32)	Initializes a new instance of the StringBuilder class from the specified substring and capacity.

Figure 7.1: Constructors for StringBuilder

Each constructor has a different list of arguments. These are known as overloads, where one member has different possible sets of arguments. The class distinguishes between the overloads to call based on the number and types of the arguments supplied.

In PowerShell, there are two different ways to use a constructor. The `New-Object` command can be used:

```
New-Object System.Text.StringBuilder
```

Or you can use the static method `new`, which was introduced with PowerShell 5:

```
[System.Text.StringBuilder]::new()
```

Static methods are explored in more detail later in this section.

In either case, and because no arguments were supplied, the first overload from *Figure 7.1* will be used. If a string were used as an argument (and it were the only argument), the third overload in the list would be used (one argument, with type `String`):

```
[System.Text.StringBuilder]::new('This is the start of the string')
```

When this constructor is used, an instance of the `StringBuilder` type is returned, which shows the properties of the class:

```
PS> [System.Text.StringBuilder]::new('This is the start of the string')
Capacity   MaxCapacity   Length
--------   -----------   ------
      31    2147483647       31
```

A property holds data about the object.

Properties

A property describes some aspect of the object. The last example shows the `Capacity`, `MaxCapacity`, and `Length` properties

of the `StringBuilder` object.

Each of the properties is described on Microsoft Docs: https://docs.microsoft.com/dotnet/api/system.text.stringbuilder?view=net-5.0#properties.

Figure 7.2 shows the properties from Microsoft Docs:

Properties

Capacity	Gets or sets the maximum number of characters that can be contained in the memory allocated by the current instance.
Chars[Int32]	Gets or sets the character at the specified character position in this instance.
Length	Gets or sets the length of the current StringBuilder object.
MaxCapacity	Gets the maximum capacity of this instance.

Figure 7.2: Properties for StringBuilder

Clicking on each property will show further detail. For example, clicking on `Capacity` will describe the property and the property's use in far more detail. The `Capacity` property documentation also includes an example that uses that property. Examples tend to be available in C#, VB, or C++. Few examples are available in PowerShell.

.NET classes also list fields as member types. Fields are indistinguishable from properties in PowerShell. In .NET, a property will implement a `get` or `set` (or both) accessor to access a value. Accessors were demonstrated in *Chapter 3, Working with Objects in PowerShell*.

The class created in the following C# snippet includes both a field and a property. The property uses `get` and `set` accessors, while the field does not:

```
Add-Type -TypeDefinition '
public class MyClass
{
    public string thisIsAField;
    public string thisIsAProperty { get; set; }
}
'
```

Once the `MyClass` type has been added to the PowerShell session, you can create an instance and use `Get-Member` to show the members of the class:

```
PS> [MyClass]::new() | Get-Member
   TypeName: MyClass
Name            MemberType Definition
----            ---------- ----------
Equals          Method     bool Equals(System.Object obj)
GetHashCode     Method     int GetHashCode()
GetType         Method     type GetType()
ToString        Method     string ToString()
thisIsAField    Property   string thisIsAField {get;set;}
thisIsAProperty Property   string thisIsAProperty {get;set;}
```

It is possible to show that the `thisIsAField` member is indeed a field by using the `GetMembers` method of the type:

```
PS> [MyClass].GetMembers() | Select-Object Name, MemberType
Name                    MemberType
----                    ----------
get_thisIsAProperty         Method
set_thisIsAProperty         Method
GetType                     Method
ToString                    Method
Equals                      Method
GetHashCode                 Method
.ctor                  Constructor
thisIsAProperty           Property
thisIsAField                 Field
```

That this is a field does not really matter to PowerShell; it can be used in the same way as a property.

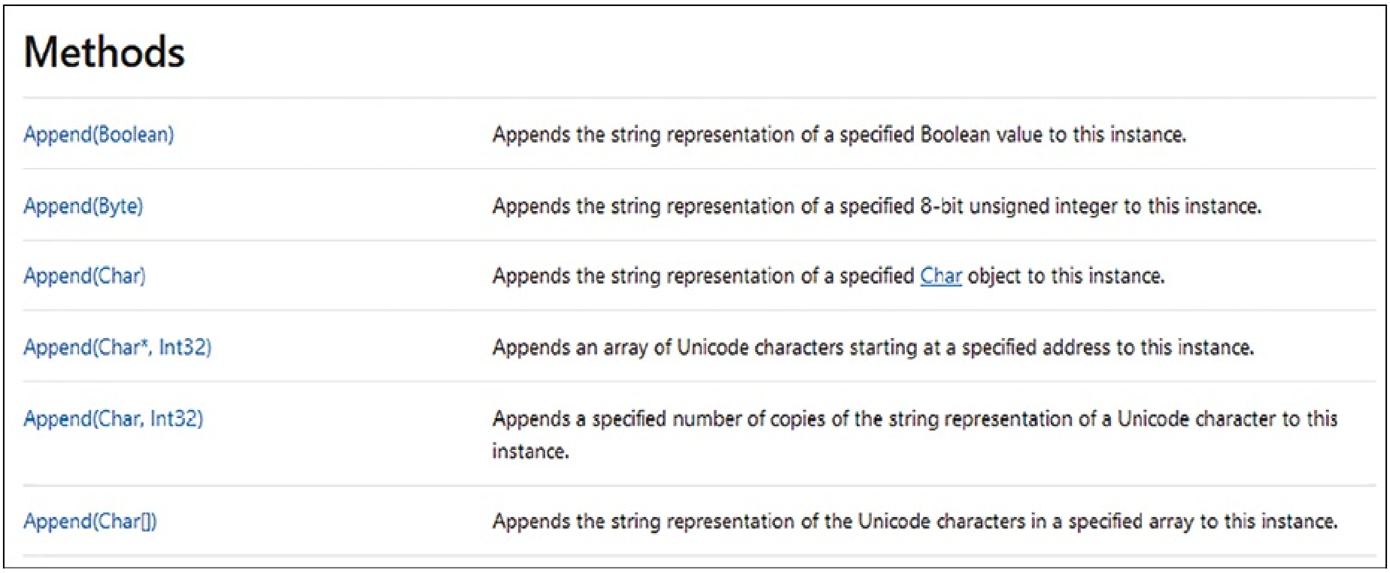
To continue to add to the string in `StringBuilder`, the methods on the class can be used.

Methods

A method enacts some change on an object. Methods can be used to change the internal state of an object, such as the `Append` method in the `StringBuilder` type. Or methods can be used to return something different from the object, such as the `ToString` method.

The methods available to the `StringBuilder` type are documented after the properties on Microsoft Docs: https://docs.microsoft.com/dotnet/api/system.text.stringbuilder?view=net-5.0#methods.

Figure 7.3 shows the first few methods:



Methods

Append(Boolean)	Appends the string representation of a specified Boolean value to this instance.
Append(Byte)	Appends the string representation of a specified 8-bit unsigned integer to this instance.
Append(Char)	Appends the string representation of a specified Char object to this instance.
Append(Char*, Int32)	Appends an array of Unicode characters starting at a specified address to this instance.
Append(Char, Int32)	Appends a specified number of copies of the string representation of a Unicode character to this instance.
Append(Char[])	Appends the string representation of the Unicode characters in a specified array to this instance.

Figure 7.3: Methods for StringBuilder

Several of the preceding methods have the same name but different arguments. As with the constructor, these methods with the same name are *overloaded*. The method that will be used is based on the number and types of the arguments.

For example, running the following will use the first of the methods in *Figure 7.3*:

```
$stringBuilder = [System.Text.StringBuilder]::new()
$stringBuilder.Append($true)
```

Running the same method name with a `Byte` will use the second method:

```
$stringBuilder.Append([Byte]1)
```

In the case of the `StringBuilder` type, each of the methods used to append to the string returns an instance of the same `StringBuilder` type.

One possible strategy for dealing with this is to pipe each statement to `Out-Null`. If the output from the method call is not interesting, then this is a perfectly valid approach:

```
$stringBuilder = [System.Text.StringBuilder]::new()
$stringBuilder.Append('Hello') | Out-Null
$stringBuilder.AppendLine() | Out-Null
$stringBuilder.AppendLine('World') | Out-Null
```

Once the string is complete, the `ToString` method may be called to show the final string:

```
PS> $stringBuilder.ToString()
Hello
World
```

If a method name is used without brackets at the end, PowerShell will show the overloads for that method instead of executing the method:

```
PS> $stringBuilder.ToString
OverloadDefinitions
-------------------
string ToString()
string ToString(int startIndex, int length)
```

Methods returning the instance of the type, as shown by `StringBuilder`, are known as fluent interfaces.

Fluent interfaces

A fluent interface is a particular pattern where each method call returns the instance of the object the method is affecting.

Fluent interfaces are intended to allow a sequence of methods to be called in turn to build a complex statement that is still easy for a human to read.

In the `StringBuilder` type this can be used to assemble a relatively complex string:

```
$string = [System.Text.StringBuilder]::new().
    AppendLine('Hello').
    AppendLine('World').
    ToString()
```

The preceding example is a simple one; however, you can use `StringBuilder` to build far more complex and elaborate strings. PowerShell itself includes another example of a fluent interface:

```
[PowerShell]::Create().
    AddCommand('Get-Process').
    AddCommand('Where-Object').
    AddParameter('Property', 'Name').
    AddParameter('Value', 'pwsh').
    AddParameter('EQ', $true).
    Invoke()
```

The preceding snippet creates a PowerShell runspace, then runs a command in that runspace. It is equivalent to the following command:

```
Get-Process | Where-Object -Property Name -Value pwsh -EQ
```

PowerShell runspaces are explored in *Chapter 15, Asynchronous Processing*.

In the previous example, `Create` is a static method. The example before used `new`, which is also a static method; however, the `new` method is added by PowerShell itself.

Static methods

Static methods can be used without creating an instance of the object. Static methods are used in a wide variety of different contexts.

`Get-Member` can be used to explore the static methods of a type in PowerShell, for example, the static methods on the `DateTime` type:

```
[DateTime] | Get-Member -MemberType Method -Static
```

`IsLeapYear` is one of the static methods on the `System.DateTime` type. It will return true when the year used as an argument is a leap year:

```
PS> [DateTime]::IsLeapYear(2020)
True
```

Another example of a static method is the `Reverse` method on the `System.Array` type. This method is notable because it does not return anything. It acts on an existing instance of an array:

```
$array = 1, 2, 3
[Array]::Reverse($array)
```

The change is directly applied to the array referenced by the `$array` variable. The variable's content will be reversed:

```
PS> $array
3
2
1
```

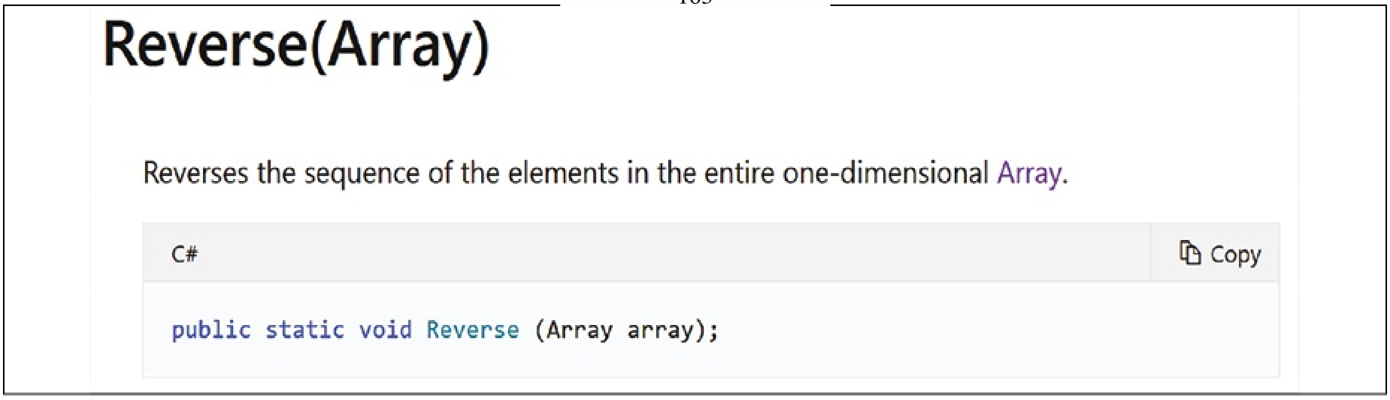
The list of methods on Microsoft Docs does not distinguish between static and non-static methods. For example, the `Reverse` method from the Methods section in Microsoft Docs is shown in *Figure 7.4*:

Reverse(Array)	Reverses the sequence of the elements in the entire one-dimensional Array.
Reverse(Array, Int32, Int32)	Reverses the sequence of a subset of the elements in the one-dimensional Array.
Reverse<T>(T[])	Reverses the sequence of the elements in the one-dimensional generic array.
Reverse<T>(T[], Int32, Int32)	Reverses the sequence of a subset of the elements in the one-dimensional generic array.

Figure 7.4: Reverse method of System.Array

Clicking into the documentation for a specific overload, for example, the first in the list above, shows that the method is static: https://docs.microsoft.com/dotnet/api/system.array.reverse?view=net-5.0#System_Array_Reverse_System_Array_.

This is shown in *Figure 7.5*:


Reverse(Array)

Reverses the sequence of the elements in the entire one-dimensional Array.

C#   Copy

```
public static void Reverse (Array array);
```

Figure 7.5: Reverse method of System.Array

Note the `static` keyword in *Figure 7.5*.

The `new` method used in a few of the examples in this chapter was added in PowerShell 5 and will not appear in the documentation on Microsoft Docs.

About the new method

The `new` static method is added to .NET types by PowerShell. The method is visible when using `Get-Member`. Because PowerShell adds the method, it does not show in Microsoft Docs.

The `new` method can be used to use any of the constructors of a class. The `new` method was used when exploring constructors to create an instance of a `StringBuilder`:

```
[System.Text.StringBuilder]::new()
```

Omitting the brackets from the end of the static method will show the overloads instead of creating the object:

```
PS> [System.Text.StringBuilder]::new
OverloadDefinitions
-------------------
System.Text.StringBuilder new()
System.Text.StringBuilder new(int capacity)
System.Text.StringBuilder new(string value)
System.Text.StringBuilder new(string value, int capacity)
System.Text.StringBuilder new(string value, int startIndex, int length, ...
System.Text.StringBuilder new(int capacity, int maxCapacity)
```

While the previous list is not as detailed as the documentation on Microsoft Docs, it can serve as a useful reminder in the console.

Properties can also be static as well as methods.

Static properties

Static properties are used to return values that are related to the type, but do not require an instance of the type to be created.

Like static methods, static properties can be listed using the `Get-Member` command:

```
PS> [DateTime] | Get-Member -MemberType Property -Static
   TypeName: System.DateTime
Name      MemberType Definition
----      ---------- ----------
MaxValue  Property   static datetime MaxValue {get;}
MinValue  Property   static datetime MinValue {get;}
Now       Property   datetime Now {get;}
Today     Property   datetime Today {get;}
UnixEpoch Property   static datetime UnixEpoch {get;}
UtcNow    Property   datetime UtcNow {get;}
```

Like static methods, Microsoft Docs does not differentiate between static and non-static properties in the property list, only in the detail of a specific property as shown in *Figure 7.6*:



Figure 7.6: System.DateTime Now static property

Static properties are used as shown here:

```
[DateTime]::Now
```

The preceding property is like using the `Get-Date` command with no parameters.

The types and members used above are public, that is, they are accessible outside of the assembly they were defined in.

Reflection in PowerShell

Types, properties, and methods can be marked as internal or private. Internal types and members are only accessible by other types and members within the same assembly. Private members are only accessible within the same class or type. Collectively these can be described as non-public types and members.

Non-public types and members are accessible using what is known as Reflection. Microsoft Docs describes Reflection in the dynamic programming guide: https://docs.microsoft.com/dotnet/framework/reflection-and-codedom/reflection.

PowerShell can make use of Reflection to explore and use non-public types and members. Doing so introduces a risk; such types and members are not part of a public-supported API. As code is updated, these non-public types and members may disappear or change behavior. It is therefore not recommended to make code that is to be used in production dependent on a non-public implementation.

Even if it might not be supported, exploring is interesting and the ability to use Reflection in PowerShell is handy.

Next up are type accelerators in PowerShell, which are described by a non-public type.

The TypeAccelerators type

The `TypeAccelators` type is used to list and add type accelerators to the PowerShell session.

The `TypeAccelerators` type is not public and therefore cannot be so easily accessed. The type may be found by using the `GetType` method on an `Assembly` object. The type is part of the `System.Management.Automation` namespace. The following command uses the `PowerShell` type to get the assembly:

```
[PowerShell].Assembly
```

The `Assembly` object may be used to run the `GetType` method, which will find a type within an assembly:

```
PS> [PowerShell].Assembly.GetType(
>>     'System.Management.Automation.TypeAccelerators'
>> )
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
False    False    TypeAccelerators                         System.Object
```

The name used with the `GetType` method is case-sensitive; the type name must be written exactly as above.

The `IsPublic` property indicates this type is not public and therefore cannot be accessed by putting the name in square brackets.

You can use the `GetType()` method on the assembly object to list all of the types within that assembly.

The type can be assigned to a variable to make it easier to work with:

```
$typeAccelerators = [PowerShell].Assembly.GetType(
    'System.Management.Automation.TypeAccelerators'
)
```

The `TypeAccelerators` type has two relevant public methods, and one relevant public property. These members, along with the `Equals` and `ReferenceEquals` methods inherited from `System.Object`, are shown in the following code:

```
PS> $typeAccelerators | Get-Member -Static
   TypeName: System.Management.Automation.TypeAccelerators
Name            MemberType Definition
----            ---------- ----------
Add             Method     static void Add(string typeName, type type)
Equals          Method     static bool Equals(System.Object objA, System...
ReferenceEquals Method     static bool ReferenceEquals(System.Object obj...
Remove          Method     static bool Remove(string typeName)
Get             Property   System.Collections.Generic.Dictionary[string,...
```

As the property and the two methods are static they are accessed using the static member operator, `::`. For example, the `Get` method can be used to list the type accelerators:

```
$typeAccelerators::Get
```

A new type accelerator can be added with the `Add` method. In the following example, a type accelerator is added to make accessing this `TypeAccelerators` type easier:

```
$typeAccelerators::Add(
    'TypeAccelerators',
    $typeAccelerators
)
```

Once done, the new type accelerator can be used in place of the variable:

```
[TypeAccelerators]::Get
```

The newly added type accelerator will go away when PowerShell is closed.

The properties and methods used on the `TypeAccelators` type are public and are therefore easy to use once an instance of the type is available. Non-public members may also be listed and used with Reflection.

When a variable is assigned a type in PowerShell, an attribute is created to handle conversions when values are assigned.

The ArgumentTypeConverterAttribute type

The `ArgumentTypeConverterAttribute` attribute is used by PowerShell when casting a variable value. The attribute is added to a variable when the type is on the left-hand side of an assignment.

For example, any values assigned to the following variable will be cast to a string:

```
[string]$variable = 'value'
```

Once the variable has been created, the presence of the attribute can be seen using `Get-Variable`, but the type it casts to is not visible:

```
PS> (Get-Variable variable).Attributes | Format-List
TransformNullOptionalParameters : True
TypeId                          : System.Management.Automation.ArgumentT...
```

Discovering the type used by the variable means exploring the non-public members of the attribute.

The attribute type itself is non-public, but an existing variable can be used to get an instance of the type itself. In the following example, `[0]` is used to access the attribute; the `Attributes` property is a collection (a `PSVariableAttributeCollection`):

```
[string]$variable = 'value'
$typeConverter = (Get-Variable variable).Attributes[0]
$typeConverterType = $typeConverter.GetType()
```

The type the attribute converts values to is held in a non-public member of the attribute. The possible members, both public and non-public, can be listed using the `GetMembers` method on the type. Without any arguments, `GetMembers` will only show public members, the same information that the `Get-Member` command shows:

```
PS> $typeConverterType.GetMembers() |
>>     Select-Object Name, MemberType, IsPublic
```

You can see the non-public members by using an overload for the `GetMembers` method. The overload accepts a `System.Reflection.BindingFlags` value. As a `Flags` enumeration, `BindingFlags` allows more than one value to be used. The values are supplied as a comma-separated string in the following example:

```
PS> $typeConverterType.GetMembers('NonPublic,Instance') |
>>     Select-Object Name, MemberType, IsPublic
Name                                      MemberType IsPublic
----                                      ---------- --------
get_TargetType                                Method    False
Transform                                     Method    False
TransformInternal                             Method    False
MemberwiseClone                               Method    False
Finalize                                      Method    False
.ctor                                    Constructor    False
TargetType                                  Property
_convertTypes                                  Field    False
```

The output from the command shows two members, which may show the target type of any value assigned to the variable. The property `TargetType`, however, looks the most promising.

The `Instance` flag used in the previous example indicates that members that are present on instances of the type should be returned. If the flag `Static` were used instead, only static members would be returned. If `Static` was used in addition to `Instance`, both static and non-static members would be displayed.

The following example shows the impact of changing the flag to `Static`:

```
PS> $typeConverterType.GetMembers('NonPublic, Static') |
>>      Select-Object Name, MemberType, IsPublic, IsStatic
Name                                       MemberType IsPublic IsStatic
----                                       ---------- -------- --------
CheckBoolValue                                 Method    False     True
ThrowPSInvalidBooleanArgumentCastException     Method    False     True
```

The `IsPublic` property is empty for the `TargetType` property. Properties allow more complex definitions of when the value can be read or written. The following snippet gets the property definition, then shows whether the `Get` method for the property is public or not:

```
PS> $typeConverterType.GetProperty(
>>     'TargetType',
>>     [System.Reflection.BindingFlags]'Instance,NonPublic'
>> ).GetMethod | Select-Object Name, IsPublic
Name           IsPublic
----           --------
get_TargetType    False
```

You can call the `GetValue` method on the preceding property definition to get the value. When calling the `GetValue` method, the instance of the attribute from `Get-Variable` must be passed as an argument. The original variable definition is included in the following to make the example complete:

```
[string]$variable = 'value'
$typeConverter = (Get-Variable variable).Attributes[0]
$typeConverterType = $typeConverter.GetType()
$targetTypeProperty = $typeConverterType.GetProperty(
    'TargetType',
    [System.Reflection.BindingFlags]'Instance,NonPublic'
)
$targetTypeProperty.GetValue($typeConverter)
```

The last line returns the type values assigned to the variable:

```
PS> $targetTypeProperty.GetValue($typeConverter)
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     String                                   System.Object
```

Reflection, as demonstrated, is complex even when the scope is as small as this single property.

You can use the `ImpliedReflection` module to simplify the process. Once the module is installed, enable it by using the `Enable-ImpliedReflection` command:

```
Install-Module ImpliedReflection
Enable-ImpliedReflection
```

A confirmation message will be displayed, which should be accepted to enable the module's functionality. Once enabled, the `TargetType` property will display on the attribute via `Get-Variable` as the following shows:

```
PS> [string]$variable = 'value'
PS> (Get-Variable variable).Attributes[0] | Format-List
TransformNullOptionalParameters : True
TypeId                          : System.Management.Automation.ArgumentT...
_convertTypes                   : {System.String}
TargetType                      : System.String
```

`ImpliedReflection` can be disabled again either by running the `Disable-ImpliedReflection` command or restarting the PowerShell console.

Summary

Delving into .NET significantly increases the flexibility of PowerShell over using built-in commands and operators. .NET is made up of hundreds of classes and enumerations, many of which can be easily used in PowerShell.

.NET types are arranged in namespaces, grouping types with similar purposes together. For example, the `System.Data.SqlClient` namespace contains types for connecting to and querying Microsoft SQL Server instances.

The `using` keyword, introduced with PowerShell 5.1, allows types to be used by name only, instead of the full name that includes the namespace.

Type accelerators have been in PowerShell since its release. PowerShell provides many built-in type accelerators allowing types to be used by a short name. Examples include `Xml` for `System.Xml.XmlDocument`, and `ADSI` for `System.DirectoryServices.DirectoryEntry`.

Types in .NET have members including constructors, properties, and methods. These are used to hold information or enact change on an object.

Static methods and properties are used throughout .NET to expose methods associated with a type, for example, the `Reverse` method of `System.Array`, or data that may be referenced, such as the `Now` property of `DateTime`.

Reflection is an advanced set of types and methods used to access information about types and members of types. Reflection exposes access to non-public members and types. While working with non-public members and types is not necessarily suitable for production code, Reflection remains a valuable tool for exploring both the inner workings of PowerShell and .NET in general.

The next chapter shows you how to work with different data types.

8

Strings, Numbers, and Dates

Access to .NET Framework means that PowerShell comes with a wide variety of ways to work with simple data types, such as strings and numbers.

This chapter covers the following topics:

- Manipulating strings
- Converting strings
- Manipulating numbers
- Manipulating dates and times

Manipulating strings, numbers, and dates have many uses, including data cleansing, retrieving specific information, and applying calculations.

Manipulating strings

The .NET `System.String` type offers a wide array of methods for manipulating or inspecting strings. The following methods are case-sensitive but are, in many cases, faster alternatives to using regular expressions, for situations when the time it takes for a script to run is important.

Working with data held in strings is an important part of any scripting language. The following sections explore selecting parts of a string, splitting, replacing, trimming, inserting, removing, and more.

Indexing into strings

In PowerShell, it is possible to access individual characters in a string by index (the zero-based position of a character) in the same way that you would access array elements using an index. Consider the following example:

```
$myString = 'abcdefghijklmnopqrstuvwxyz'
$myString[0]     # This is a (the first character in the string)
$myString[-1]    # This is z (the last character in the string)
```

Ranges of characters can be selected from the string by using a range or a list of indexes. The resulting array of characters can be joined with the `-join` operator. For example:

```
$myString = 'abcdefghi'
-join $myString[0..5]
```

The preceding example returns `abcdef`, the first 6 characters (indexes 0 to 5, inclusive) of the original string.

The `System.String` type includes several methods that can be used to manipulate strings. The preceding operation can be performed using the `Substring` method.

Substring

The `Substring` method allows part of a string to be taken, either all characters starting from a certain index or a specific number of characters from an index.

Having two different possible sets of arguments for a single method name is known as overloading. You can view the two

different possible sets of arguments by running the method name without parentheses:

```
PS> 'anyString'.Substring
OverloadDefinitions
-------------------
string Substring(int startIndex)
string Substring(int startIndex, int length)
```

The following example selects everything from index 20 to the end of the string:

```
$myString = 'abcdefghijklmnopqrstuvwxyz'
$myString.Substring(20)   # Start at index 20. Returns 'uvwxyz'
```

The second overload for `Substring` selects a specific number of characters from a starting point:

```
$myString = 'abcdefghijklmnopqrstuvwxyz'
$myString.Substring(3, 4) # Start at index 3, get 4 characters.
```

The index starts at 0, counting from the beginning of the string.

The `Substring` method raises an error if the `length` argument is longer than the string:

```
PS> $myString = 'abcde'
PS> $myString.Substring(0, 6)
MethodInvocationException: Exception calling "Substring" with "1" argument(s): "startIndex cannot be
larger than length of string. (Parameter 'startIndex')"
```

When changing a set of strings, each with a different length, there are several ways to avoid this problem. A condition may be added:

```
$myString = 'abcdefghi'
if ($myString.Length -gt 6) {
    $myString.Substring(0, 6)
}
```

Or the smallest of two numbers might be selected by using the `Min` method in the `Math` class. The `Min` method chooses the smallest of two numeric arguments:

```
$myString = 'abcdefghi'
$myString.Substring(0, [Math]::Min(6, $myString.Length))
```

Split

The `Split` method has a relative in PowerShell: the `-split` operator. The `-split` operator expects a regular expression, whereas the `Split` method splits using a character, an array of characters, a string, or an array of strings.

The following example splits based on a comma:

```
$myString = 'Surname,GivenName'
$myString.Split(',')
```

When splitting the following string based on a comma, the resulting array will have three elements. The first element is `Surname`, the last is `GivenName`. The second element in the array (index 1) is blank:

```
$string = 'Surname,,GivenName'
$array = $string.Split(',')
$array.Count     # This is 3
$array[1]        # This is empty
```

This blank value may be discarded by setting the `StringSplitOptions` argument of the `Split` method:

```
$string = 'Surname,,GivenName'
$array = $string.Split(
    ',',
    [StringSplitOptions]::RemoveEmptyEntries
)
$array.Count     # This is 2
```

When using the `Split` method in this manner, individual variables may be filled from each value as follows:

```
$surname, $givenName = $string.Split(
    ',',
    [StringSplitOptions]::RemoveEmptyEntries
)
```

The `Split` method is powerful, but care is required when using its different arguments. Each of the different sets of arguments works as follows:

```
PS> 'string'.Split
OverloadDefinitions
-------------------
string[] Split(char separator, System.StringSplitOptions options)
string[] Split(char separator, int count, System.StringSplitOptions options)
string[] Split(Params char[] separator)
string[] Split(char[] separator, int count)
string[] Split(char[] separator, System.StringSplitOptions options)
string[] Split(char[] separator, int count, System.StringSplitOptions options)
string[] Split(string separator, System.StringSplitOptions options)
string[] Split(string separator, int count, System.StringSplitOptions options)
string[] Split(string[] separator, System.StringSplitOptions options)
string[] Split(string[] separator, int count, System.StringSplitOptions options)
```

In PowerShell, it is not always entirely clear which of these methods will be chosen when running the following example:

```
$string = 'one||two||three'
$string.Split('||')
```

In PowerShell 7, the argument `||` is treated as a string. The result is therefore the values `one`, `two`, and `three`.

In Windows PowerShell, the argument `||` is treated as an array of characters. The result is `one`, an empty string, `two`, an empty string, and `three`.

The difference between versions can be problematic if code using the method is expected to work in both versions of PowerShell. The behavior can be made consistent by using one overload that accepts `string[]` and `StringSplitOptions`. For example, the following version will behave in the same way in both PowerShell and Windows PowerShell:

```
$string = 'one||two||three'
$string.Split([string[]]'||', [StringSplitOptions]::None)
```

Replace

The `Replace` method will substitute one string value for another:

```
$string = 'This is the first example'
$string.Replace('first', 'second')
```

PowerShell also has a `-replace` operator. The `-replace` operator uses a regular expression to describe the value that should be replaced.

Regular expressions (discussed in *Chapter 9*, *Regular Expressions*) may be more difficult to work with in some cases, especially when replacing characters that are reserved in regular expressions (such as the period character, `.`, which matches any single character). The period character is escaped with `\` to make the two replacements give the same result:

```
$string = 'Begin the begin.'
$string -replace 'begin\.', 'story, please.'
$string.Replace('begin.', 'story, please.')
```

The `-replace` operator can be replaced with `-creplace` to enforce case-sensitivity.

Conversely, the `Replace` method is case-sensitive by default. The following replacement has no effect on the string:

```
PS> 'begin'.Replace('B', '')
begin
```

Extra arguments can be used to change the default behavior, using one of the following two overloads:

```
string Replace(string oldValue, string newValue, bool ignoreCase, cultureinfo culture)
string Replace(string oldValue, string newValue, System.StringComparison comparisonType)
```

Either of the following approaches can be used:

```
'begin'.Replace('B', '', $true, (Get-Culture))
'begin'.Replace('B', '', [System.StringComparison]::OrdinalIgnoreCase)
```

The value for the `cultureInfo` argument is obtained using the `Get-Culture` command in the first line.

The second line uses an `OrdinalIgnoreCase` comparison, one of the possible values from the `StringComparison` enumeration documented on Microsoft Docs:

https://docs.microsoft.com/dotnet/api/system.stringcomparison?view=net-5.0

Trim, TrimStart, and TrimEnd

The `Trim` method, by default, removes all white space (spaces, tabs, and line breaks) from the beginning and end of a string. Consider the following example:

```
$string = "
    This string has leading and trailing white space      "
$string.Length
$string.Trim().Length
```

Before `Trim` runs, the string is 60 characters long. After `Trim` has been used, the string length is 48 characters.

The `TrimStart` and `TrimEnd` methods limit their operation to either the start or end of the string.

Each of the methods accepts a list of characters to trim. Consider the following example:

```
$string = '*__This string is surrounded by clutter.--#'
$string.Trim('*_-#')
```

The `Trim` method does not remove a string from the end of another. The string supplied in the previous example (`'*_-#'`) is treated as an array. This can be seen in the definition of the method:

```
PS> 'string'.Trim
OverloadDefinitions
-------------------
string Trim(Params char[] trimChars)
string Trim()
```

A failure to appreciate this can lead to unexpected behavior. The domain name in the following example ends with the suffix `'.uk.net'` . The goal is to trim the suffix from the end of the string. However, the method goes too far here, taking away part of the name:

```
PS> $string = 'magnet.uk.net'
PS> $string.TrimEnd('.uk.net')
mag
```

Trimming a string removes characters from the start and end; it is sometimes necessary to add or remove content from the middle of the string.

Insert and Remove

The `Insert` method adds one string into another. This method expects an index from the beginning of the string, counting from 0, and a string to insert, as follows:

```
$string = 'The letter of the alphabet is a'
$string.Insert(4, 'first ')  # Insert this before "letter", include a trailing space
```

The `Remove` method removes characters from a string, based on a start position and the length of the string to remove:

```
$string = 'This is is an example'
$string.Remove(4, 3)
```

The previous statement removes the first instance of `is` , including the trailing space.

IndexOf and LastIndexOf

`IndexOf` and `LastIndexOf` may be used to locate a character or string within a string. `IndexOf` finds the first occurrence of a string, and `LastIndexOf` finds the last occurrence of the string. In both cases, the zero-based index of the start of the string is returned. If the character, or string, isn't present, the two methods will return `-1` :

```
$string = 'abcdefedcba'
$string.IndexOf('b')     # Returns 1
$string.LastIndexOf('b') # Returns 9
$string.IndexOf('ed')    # Returns 6
```

As `-1` is used to indicate that the value is absent, the method is not suitable for statements based on an implicit Boolean. The index 0, a valid position, would be considered false. The following example correctly handles the return value from `IndexOf` in a conditional statement:

```
$string  = 'abcdef'
if ($string.IndexOf('a') -gt -1) {
     'The string contains an a'
}
```

The scope of the `IndexOf` and `LastIndexOf` methods can be limited using the start index and count arguments.

Methods that can locate a position within a string are useful when combined with other string methods, as shown here:

```
PS> $string = 'First,Second,Third'
PS> $string.Substring(
>>     $string.IndexOf(',') + 1, # startIndex (6)
>>     $string.LastIndexOf(',') - $string.IndexOf(',') - 1 # length (6)
>> )
Second
```

The `IndexOf` and `LastIndexOf` methods are frequently used alongside other string methods, including `Substring`, `Insert`, and `Remove`.

PadLeft and PadRight

The `PadLeft` and `PadRight` options endeavor to increase the length of a string up to a given maximum length. Both `PadLeft` and `PadRight` take the same arguments, as follows:

```
PS> ''.PadRight
OverloadDefinitions
-------------------
string PadRight(int totalWidth)
string PadRight(int totalWidth, char paddingChar)
PS> ''.PadLeft
OverloadDefinitions
-------------------
string PadLeft(int totalWidth)
string PadLeft(int totalWidth, char paddingChar)
```

Both methods make a new string up to the total width. If the string is already equal to or longer than the total width, it will not be changed. By default, the padding character is a space.

The following example pads the right-hand side of strings, using "." as the padding character argument:

```
PS> ('one', 'two', 'three').PadRight(10, '.')
one.......
two.......
three.....
```

Padding a string on the left, in effect, aligns the string on the right:

```
PS> ('one', 'two', 'three').PadLeft(10, '.')
.......one
.......two
.....three
```

If a string value is being padded with spaces, the format operator can be used instead:

```
PS> '"{0,10}"' -f 'one'     # Pad left
"       one"
PS> '"{0,-10}"' -f 'one'    # Pad right
"one       "
```

Padding and alignment are useful when generating formatted strings.

ToUpper, ToLower, and ToTitleCase

`ToUpper` converts any lowercase characters in a string into uppercase. `ToLower` converts any uppercase characters in a string into lowercase:

```
'aBc'.ToUpper()    # Returns ABC
'AbC'.ToLower()    # Returns abc
```

Considering that the methods discussed here are case-sensitive, converting a string into a known case may be an important first step. Consider the following example:

```
$string = 'AbN'
$string = $string.ToLower()
$string = $string.Replace('n', 'c')
```

`ToTitleCase` is not a method of the `String` object. It is a method of the `System.Globalization.TextInfo` class. The `ToTitleCase` method performs limited culture-specific capitalization of words:

```
PS> (Get-Culture).TextInfo.ToTitleCase('some title')
Some Title
```

As this is not a static method, the `TextInfo` object must be created first. This object cannot be directly created. `TextInfo` can be obtained via the `System.Globalization.CultureInfo` object, and this object is returned by the `Get-Culture` command.

The same `TextInfo` object may also be accessed using the host automatic variable:

```
$host.CurrentCulture.TextInfo.ToTitleCase('another title')
```

The `ToTitleCase` method does not convert words that are entirely uppercase as they are acronyms.

Contains, StartsWith, and EndsWith

The `Contains`, `StartsWith`, and `EndsWith` methods each return `true` or `false`, depending on whether the string contains, starts with, or ends with the specified string.

`Contains` returns `true` if the value is found within the subject string:

```
$string = 'I am the subject'
$string.Contains('the')    # Returns $true
```

`StartsWith` and `EndsWith` return `true` if the subject string starts or ends with the specified value:

```
$string = 'abc'
$string.StartsWith('ab')
$string.EndsWith('bc')
```

Comparisons using each of these methods are case-sensitive by default. Case-sensitivity can be controlled using the overloads for each of the methods. For example, `StartsWith` and `EndsWith` both accept a comparison type argument in the same way as used by the `Replace` method:

```
$string = 'abc'
$string.StartsWith('ab', [System.StringComparison]::OrdinalIgnoreCase)
```

The `Contains` and `EndsWith` methods support the comparison type argument used above for `StartsWith`.

Many of the preceding methods can be called on an array of strings without looping.

String methods and arrays

In PowerShell, you can call some string methods on an array. PowerShell automatically unrolls the array and executes the method. Array unrolling applies when accessing properties as well; however, this does not apply to strings.

The method will be executed against each of the elements in the array. For example, the `Trim` method is used against each of the strings as follows:

```
PS> ('azzz', 'bzzz', 'czzz').Trim('z')
a
b
c
```

The `Split` method is also capable of acting against an array, the result of the following example is a single array containing four elements:

```
PS> ('a,b', 'c,d').Split(',')
a
b
c
d
```

This remains `true` if the array object does not have a conflicting method or property. For example, the `Insert` method cannot be used as an array object has a version of its own.

Properties and methods of array elements

The array unrolling feature demonstrated here has broader scope than methods, and it applies to more than string objects.

In the case of strings, you can view the methods that can be used as follows:

```
$arrayMembers = (Get-Member -InputObject @() -MemberType Property, Method).Name
'string' |
    Get-Member -MemberType Property, Method |
    Where-Object Name -notin $arrayMembers
```

Using this feature with `DateTime` objects, the `AddDays` method may be called on each element in an array:

```
((Get-Date '01/01/2017'), (Get-Date '01/02/2017')).AddDays(5)
```

Likewise, the `DayOfWeek` property may be accessed on each element in the array, as follows:

```
((Get-Date '01/01/2017'), (Get-Date '01/02/2017')).DayOfWeek
```

A similar `Get-Member` command reveals the list of properties and methods that may be used in this manner:

```
Get-Date |
    Get-Member -MemberType Property, Method |
    Where-Object Name -notin $arrayMembers
```

Chaining methods

As many of the string methods return a string, the methods can be chained together. For example, each of the following methods returns a string, so another method can be added to the end:

```
PS> ' ONe*? '.Trim().TrimEnd('?*').ToLower().Replace('o', 'O')
One
```

Each method is executed in turn, each method returns a new string for the next method.

A series of replacements might be made by using this technique. This example is contrived, it would be easier to use `Replace`

than `Remove` and `Insert`. The approach is more useful when something else is being used to reveal position information.

The sample string has three placeholders that should be replaced with a fixed set of values:

```
$string = 'one __ three four _____ six ___'
$valuesToInsert = 'two', 'five', 'seven'
```

First the location of each value to replace is found. This step is imagined to be revealed by something else:

```
$position = 0
$positions = while ($position -lt $string.Length) {
    $positionInfo = [PSCustomObject]@{
        Start  = $start = $string.IndexOf('_', $position)
        End    = $string.IndexOf(' ', $start)
        Length = 0
    }
    if ($positionInfo.End -eq -1) {
        $positionInfo.End = $string.Length
    }
    $positionInfo.Length = $positionInfo.End - $positionInfo.Start
    $position = $positionInfo.End
    $positionInfo
}
```

Viewing the value of the `$positions` variable shows it has recorded the start, end, and length of each placeholder string:

```
PS> $positions
Start End Length
----- --- ------
    4   6      2
   18  23      5
   28  31      3
```

Then the discovered values are substituted in reverse order. Reversing the order retains the value of the position information. Inserting a string of a different length would otherwise invalidate the value of `Start`:

```
for ($i = $positions.Count - 1; $i -ge 0; $i--) {
    $positionInfo = $positions[$i]
    $string = $string.Remove(
        $positionInfo.Start,
        $positionInfo.Length
    ).Insert(
        $positionInfo.Start,
        $valuesToInsert[$i]
    )
}
```

Updating strings like this is a useful thing to be able to do. Complex replacements can be made without disturbing the overall structure of a document.

PSKoans uses Insert and Replace

I used the previous technique when writing the commands used to update user content in the PSKoans module:

https://aka.ms/PSKoans

The location information in that case is drawn from the **Abstract Syntax Tree** (**AST**) in PowerShell. AST is explored in *Chapter 21, Testing*. The result of the search is a series of positions in a file where text must be replaced with new string values.

The previous methods all work on an existing string and often return a new string. It may be desirable to convert a string to another type entirely.

Converting strings

PowerShell has a variety of commands that can be used to convert strings into objects.

Several commands are available specifically to work with strings:

- The `*-Csv` commands
- `ConvertFrom-StringData`
- `Convert-String`
- `ConvertFrom-String`

The `Convert-String` and `ConvertFrom-String` commands are part of a project called Flash Extract by Microsoft Research and have never been converted for use in .NET Core. These commands are only available in Windows operating systems and only available in PowerShell 7 via a compatibility session.

In addition to these commands, PowerShell can make use of the `ToBase64String` and `FromBase64String` methods of the `Convert` class to work with Base64-encoded data.

The *-Csv commands

`ConvertTo-Csv` turns objects in PowerShell into **comma-separated value** (**CSV**) strings:

```
PS> Get-Process -Id $pid | Select-Object Name, Id, Path | ConvertTo-Csv
"Name","Id","Path"
"pwsh","6676","C:\Program Files\PowerShell\7\pwsh.exe"
```

In the preceding example, Windows PowerShell also includes type data by default. PowerShell 7 does not include the type data line.

`ConvertFrom-Csv` turns CSV-formatted strings into a custom object:

```
PS> "David,0123456789,28" | ConvertFrom-Csv -Header Name, Phone, Age
Name  Phone      Age
----  -----      ---
David 0123456789 28
```

As `ConvertFrom-Csv` is specifically written to read CSV-formatted data, it discards quotes that surround strings but allows fields to spread across lines and so on. Consider the following example:

```
'David,0123456789,28,"1 Some street, A Lane"' | ConvertFrom-Csv -Header Name, Phone, Age, Address |
Format-Table -Wrap
```

If the `Header` parameter is not defined, the first line read by `ConvertFrom-Csv` is expected to be a header. If there is only one line of data, then nothing is returned:

```
'Name,Age', 'David,28' | ConvertFrom-Csv
```

`Export-Csv` and `Import-Csv` complement these two commands by writing and reading information to a file instead:

```
Get-Process -Id $pid |
    Select-Object Name, Id, Path |
    Export-Csv 'somefile.csv'
Import-Csv somefile.csv
```

ConvertFrom-StringData

The `ConvertFrom-StringData` command can be used to convert key and value pairs written in a string into a `Hashtable`. `ConvertFrom-StringData` is useful when working with paired values (where each pair is on a separate line).

The key and value pairs must be separated by the `=` character for `ConvertFrom-StringData`. However, replacing another separator is a small step.

For example, the following string can be converted to an object:

```
$string = @"
Name : John Doe
Username : jdoe
"@
$string.Replace(':', '=') | ConvertFrom-StringData
```

The preceding command will return a single Hashtable. White space before and after the key or value is discarded.

The resulting Hashtable can potentially be converted into a custom object:

```
$string = @"
Name : John Doe
Username : jdoe
"@
[PSCustomObject]($string.Replace(':', '=') | ConvertFrom-StringData)
```

If the lines to convert are separate elements in an array the command will create one Hashtable for each line:

```
$string = @(
    'Name : John Doe'
    'Username : jdoe'
)
$hashtables = $string.Replace(':', '=') | ConvertFrom-StringData
$hashtables.GetType()
```

The preceding command shows the output is `object[]`, and in this case an array of Hashtables. If using `Get-Content` to read the original string, the `-Raw` parameter can be used to read the value as a single string.

`Out-String` may be used to work with arrays of strings from other sources:

```
$string = @(
    'Name : John Doe'
    'Username : jdoe'
) | Out-String
[PSCustomObject]($string.Replace(':', '=') | ConvertFrom-StringData)
```

Convert-String

The `Convert-String` command may be used to simplify some string conversion operations. `Convert-String` is not included in PowerShell 7 and therefore is not present on Linux and macOS installations.

The conversion is performed based on an example that must be supplied. For example, `Convert-String` can generate account names from a list of users:

```
'Michael Caine', 'Benny Hill', 'Raf Vallone' |
    Convert-String -Example 'Michael Caine=MCaine'
```

The `Example` parameter uses the generalized syntax as follows:

```
<Before>=<After>
```

This example text does not have to be one of the sets being converted, it only must show the existing format (`Before`) and the desired format (`After`). For example, the following works:

```
PS> 'Michael Caine', 'Benny Hill', 'Raf Vallone' |
>>      Convert-String -Example 'First Second=FSecond'
MCaine
BHill
RVallone
```

The following alternate syntax is also supported, the output is the same as the version above:

```
'Michael Caine', 'Benny Hill', 'Raf Vallone' |
    Convert-String -Example @{
        Before = 'First Second'
        After  = 'FSecond'
    }
```

The `Convert-String` command is not without its limitations. `After` may only include strings, or partial strings, from `Before` , along with a subset of punctuation characters. Characters that aren't permitted in `After` include `@` , `$` , `~` , `` ` `` , and `!` . Because of these limitations, `Convert-String` cannot, for example, build an email address for each user in the list in a single step.

ConvertFrom-String

`ConvertFrom-String` can be used to convert strings into objects. `ConvertFrom-String` is part of the same module as `Convert-String` ; it is therefore only available in Windows PowerShell.

`ConvertFrom-String` performs two different styles of string to object conversion. The first behaves much as `ConvertFrom-Csv` does, except that it does not discard characters that make up the CSV format. In the following example, the quotation marks surrounding the first name are preserved:

```
PS> '"bob",tim,geoff' | ConvertFrom-String -Delimiter ',' -PropertyNames name1, name2, name3
name1 name2 name3
----- ----- -----
"bob" tim   geoff
```

The default delimiter (if the parameter is not supplied) is a space. The second operating mode of `ConvertFrom-String` is far more complex. A template must be defined for each element that is to be pushed into a property.

The following example uses `ConvertFrom-String` to convert the output from the `tasklist` command into a collection of custom objects:

```
$template = '{Task*:{ImageName:System Idle Process} {[Int]PID:0} {SessionName:Services} {Session:0}
{Memory:24 K}}'
tasklist |
    Select-Object -Skip 3 |
    ConvertFrom-String -TemplateContent $template |
    Select-Object -ExpandProperty Task
```

The `Task*` element in the template denotes the start of an object. It allows each of the remaining fields, `ImageName` , `PID` , `SessionName` , `Session` , and `Memory` to be grouped together under a single object.

An example of the output from the command is shown here:

```
ImageName   : System Idle Process
PID         : 0
SessionName : Services
Session     : 0
Memory      : 8 K
ImageName   : System
PID         : 4
SessionName : Services
Session     : 0
Memory      : 3,364 K
```

The `ConvertFrom-String` command is good at dealing with well-formatted data that is already divided correctly. In the case of the `tasklist` command, the end of a single task (or data record) is denoted by a line break.

Working with Base64

Base64 is a string encoding for arrays of bytes. The bytes can represent just about anything at all. Base64 is frequently used with network protocols to send more complex data across a network; for example, email attachments are encoded using Base64 so images and other files can be sent across a network as a string.

The `Convert` type may be used to take a Base64 string and convert it to an array of bytes. In the following example, the bytes represent an ASCII string:

```
PS> $bytes = [Convert]::FromBase64String('SGVsbG8gd29ybGQ=')
PS> -join ($bytes -as [char[]])
Hello world
```

The `ToBase64String` method accepts an array of bytes and converts those to a Base64 string. The array of bytes is provided by coercing an array of characters to an array of bytes. The conversion from `char[]` to `byte[]` is implicit when using the method:

```
[Convert]::ToBase64String([char[]]'Hello world')
```

Base64 encoding is used by the `-EncodedCommand` parameter of the PowerShell executable. This allows a complex command to be passed to PowerShell without the need for a script file.

PowerShell expects the string to be Unicode encoded; each character is represented by two bytes. The `Unicode` static property of `System.Text.Encoding` is used to convert a string in PowerShell to an array of bytes. The resulting byte array is converted to a Base64 string:

```
$script = {Write-Host 'Hello world'}
$bytes = [System.Text.Encoding]::Unicode.GetBytes($script)
[Convert]::ToBase64String($bytes)
```

The resulting string can be used with either `powershell.exe` or `pwsh.exe`:

```
pwsh -encodedcommand VwByAGkAdABlAC0ASABvAHMAdAAgACcASABlAGwAbABvACAAdwBvAHIAbABkACcA
```

Running the command will run the `Write-Host` command. The preceding example is simple, but the technique may be used with far longer and more complicated scripts. Base64 is used to encode special characters such as quotes, brackets, curly braces, and so on, so a complex command can be passed as a much simpler argument.

Manipulating numbers

Basic mathematical operations in PowerShell make use of the operators discussed in *Chapter 4, Operators*.

Formatting numbers was also introduced in *Chapter 4*, *Operators*; the following examples review the use of the format operator with numeric values:

```
'{0:x}' -f 24244      # Lower-case hexadecimal. Returns 5eb4
'{0:X}' -f 24244      # Upper-case hexadecimal. Returns 5EB4
'{0:P}' -f 0.28232    # Percentage. Returns 28.23%
'{0:N2}' -f 32583.122 # Culture specific number format.
                      # 2 decimal places.
                      # Returns 32,583.12 (in culture en-GB)
```

Format operations like those above return string values. Once a number has been formatted with the format operator, `-f`, it will no longer sort as a numeric value.

Large byte values

PowerShell provides operators for working with large byte counts. These operators are as follows:

- **nKB**: Kilobytes (*n * 1024^1*)
- **nMB**: Megabytes (*n * 1024^2*)
- **nGB**: Gigabytes (*n * 1024^3*)
- **nTB**: Terabytes (*n * 1024^4*)
- **nPB**: Petabytes (*n * 1024^5*)

Kibibyte and mebibytes

PowerShell uses the JEDEC memory standards for the meaning of KB, MB, and GB. Therefore, each of the values for KB, MB, and so on is 1,024 to some power rather than 1,000 to some power as documented in IEEE 1541-2002.

These operators can be used to easily represent large values:

```
PS> 22.5GB
24159191040
```

The operators may also be used to convert large byte values into shorter values. For example, a shorter value might be added to a message using the format operator, as shown here:

```
PS> '{0:F} TB available' -f (123156235234522 / 1TB)
112.01 TB available
```

Power of 10

PowerShell uses the `e` operator to represent a scientific notation (power-of-10, "* 10^n") that can be used to represent large numbers. The exponent can be either positive or negative:

```
2e2    # Returns 200 (2 * 10^2)
2e-1   # Returns 0.2 (2 * 10^-1)
```

Hexadecimal

Hexadecimal formats are easily accessible in PowerShell. PowerShell returns the decimal form of any given hexadecimal number when the hexadecimal value is prefixed with `0x`:

```
PS> 0x5eb4
24244
```

Using System.Math

While PowerShell itself comes with reasonably basic mathematical operators, the .NET `System.Math` type has a far wider variety.

The `System.Math` type implements each operation as a static method. The static methods of the type can be shown using the `Get-Member` command:

```
[Math] | Get-Member -Static
```

The `Round` static method of the `Math` type can be used to round up to a fixed number of decimal places. In the following example, the value is rounded to two decimal places:

```
[Math]::Round(2.123456789, 2)
```

By default, the `Round` method in .NET performs what is known as banker's rounding. It always prefers to round up to an even number. For example, 1.5 will round to 2, and 2.5 will round to 2.

Rounding behavior can be changed using the `MidpointRounding` enumeration, as the following shows:

```
# Will return 2.22
[Math]::Round(2.225, 2)
# Will return 2.23
[Math]::Round(2.225, 2, [MidpointRounding]::AwayFromZero)
```

The `Ceiling` and `Floor` methods are used when performing whole number rounding:

```
[Math]::Ceiling(2.1234)    # Returns 3
[Math]::Floor(2.9876)      # Returns 2
```

`Abs` converts a positive or negative integer into a positive integer (and multiplies by -1 if the value is negative):

```
[Math]::Abs(-45748)
```

Numbers may be raised to a power using the following syntax:

```
[Math]::Pow(2, 8) # Returns 256 (28)
```

A square root can be calculated as follows:

```
[Math]::Sqrt(9)    # Returns 3
```

The `System.Math` class also contains static properties for mathematical constants:

```
[Math]::pi    # π, 3.14159265358979
[Math]::e     # e, 2.71828182845905
```

Methods are also available to work with log, tan, sin, cos, and so on.

Converting strings into numeric values

In many cases, strings may be coerced to numeric values. Consider the following example:

```
[Int]"2"            # String to Int32
[Decimal]"3.141"    # String to Decimal
[UInt32]"10"        # String to UInt32
[SByte]"-5"         # String to SByte
```

If a value cannot be coerced, for example the string `1MB`, or the value is a hexadecimal string such as `a1b2`, then the value can be multiplied by `1` to convert the value into a number:

```
$string = '5eb4'
1 * "0x$string"
```

Or the same technique can be used for a size in GB:

```
$size = '1.21GB'
1 * $size
```

In this case, GB is executed as if `1.21GB` has been typed directly into the console.

The string may need extraneous characters removing first; for example:

```
$size = '1,024MB'
1 * ($size -replace ',') / 1GB
```

For advanced conversions, the `System.Convert` class may be used. The `Convert` class includes static methods that can take a string and convert it into a number using a specified base.

A binary (base `2`) value is converted to a 32-bit integer as follows:

```
[Convert]::ToInt32('01000111110101', 2)  # Returns 4597
```

The `Convert` type supports conversion using base 2 (binary), 8 (octal), 10 (denary), and 16 (hexadecimal).

Manipulating dates and times

`DateTime` objects may be created in several ways. The `Get-Date` command is one of these. The `DateTime` type has several static methods that can be used, and an instance of `DateTime` has methods that might be used.

Working with dates includes converting a string representing a date into a `DateTime` object or finding a date that is relative to the current date and time.

Parsing dates

The `Get-Date` command is the best first stop for converting strings into dates. `Get-Date` deals with a reasonable number of formats.

If, however, `Get-Date` is unable to help, the `DateTime` class has two static methods that can be used:

- `ParseExact`
- `TryParseExact`

The format strings used by these methods are documented on Microsoft Docs:

https://docs.microsoft.com/dotnet/standard/base-types/custom-date-and-time-format-strings

The `ParseExact` method accepts one or more format strings, and returns a `DateTime` object:

```
$string = '20170102-2030'  # Represents 1st February 2017, 20:30
[DateTime]::ParseExact($string, 'yyyyddMM-HHmm', (Get-Culture))
```

The culture returned from `Get-Culture` is used as the format provider argument for the `ParseExact` method.

The format string uses the following syntax:

- `yyyy` to represent a four-digit year
- `dd` for a two-digit day
- `MM` for a two-digit month
- `HH` for the hours in the day (this is for 24 hour format; `hh` is used for 12 hour format)

The `ParseExact` method can account for more than one date format. In this case, two variations of the format are accepted, the second of which expects seconds (`ss`):

```
$strings = '20170102-2030', '20170103-0931.24'
[String[]]$formats = 'yyyyddMM-HHmm', 'yyyyddMM-HHmm.ss'
foreach ($string in $strings) {
    [DateTime]::ParseExact($string, $formats, (Get-Culture), 'None')
}
```

The final argument, `None` , grants greater control over the parsing process. The other possible values and the effects are documented on Microsoft Docs:

https://docs.microsoft.com/dotnet/api/system.globalization.datetimestyles

If the `ParseExact` method fails, an exception is thrown.

`TryParseExact` is an alternative to `ParseExact` . Instead of throwing an error, `TryParseExact` returns a Boolean, `true` or `false` . The `TryParseExact` method itself returns `true` or `false` , depending on whether it was able to parse the string.

The parsed date can be extracted using a reference to an existing date variable. The method updates the value held in the variable if parsing is successful, as follows:

```
$date = Get-Date 01/01/1601   # A valid DateTime object with an obvious date
$string = '20170102-2030'
if ([DateTime]::TryParseExact($string, 'yyyyddMM-HHmm', $null, 'None', [Ref]$date)) {
    $date
}
```

You use the `[Ref]` type, a type accelerator for `System.Management.Automation.PSReference` , as a value that `TryParseExact` can change. This allows the method to return `true` as well as setting a variable value to the parsed date. If `TryParseExact` returns `false` , the date in the variable is not changed.

Changing dates

You can change a `DateTime` object in several ways.

A `Timespan` object can be added to or subtracted from a date:

```
(Get-Date) + (New-Timespan -Hours 6)
```

The `Date` property can be used, representing the start of the day:

```
(Get-Date).Date
```

You can use the `Add<Interval>` methods to add and subtract time, as follows:

```
(Get-Date).AddDays(1)  # One day from now
(Get-Date).AddDays(-1) # One day before now
```

Adding days to a date like this is frequently used to search a filesystem. For example, this is how to search for files edited in the last 7 days:

```
Get-ChildItem -File |
    Where-Object LastWriteTime -gt (Get-Date).AddDays(-7)
```

You can use a boundary by defining a start and end time for a comparison, in this case, to find all files modified between midnight 7 days ago and midnight last night:

```
$start = (Get-Date).Date.AddDays(-7)
$end = (Get-Date).Date
Get-ChildItem -File | Where-Object {
    $_.LastWriteTime -gt $start -and
    $_.LastWriteTime -le $end
}
```

In addition to `AddDays`, the `DateTime` object makes the following available:

```
(Get-Date).AddTicks(1)
(Get-Date).AddMilliseconds(1)
(Get-Date).AddSeconds(1)
(Get-Date).AddMinutes(1)
(Get-Date).AddHours(1)
(Get-Date).AddMonths(1)
(Get-Date).AddYears(1)
```

By default, dates returned by `Get-Date` are local (that is, within the context of the current time zone). A date may be converted into UTC as follows:

```
(Get-Date).ToUniversalTime()
```

The `ToUniversalTime` method only changes the date if the `Kind` property of the date is set to `Local` or `Unspecified`. This is shown in the following snippet:

```
PS> Get-Date | Select-Object DateTime, Kind
DateTime                    Kind
--------                    ----
30 October 2018 18:38:41    Local
```

The `ToLocalTime` method adjusts the date in accordance with the system's current time zone. This operation is performed if `Kind` is `Utc` or unspecified.

A date of a specific `Kind` may be created as follows, enabling the appropriate use of `ToLocalTime` or `ToUniversalTime`. When creating a date using the constructor for `DateTime`, the time must be provided in ticks; the number of 100-nanosecond intervals since the first day of 1601:

```
$UtcDate = [DateTime]::new((Get-Date).Ticks, 'Utc')
```

Dates may be converted into a string, either immediately using `Get-Date` with the `Format` parameter or using the `ToString` method. The `Format` parameter and `ToString` method accept the same arguments.

The date strings created by the following statements are equal:

```
Get-Date -Format 'dd/MM/yyyy HH:mm'
(Get-Date).ToString('dd/MM/yyyy HH:mm')
```

The `ToString` method is useful, as it allows you to adjust a date by chaining properties and methods before converting it into a string. The following command finds midnight (in universal time) one week before today's date and then converts it to a string:

```
(Get-Date).ToUniversalTime().
    Date.
    AddDays(-7).
    ToString('dd/MM/yyyy HH:mm')
```

The `ToString` method used on `DateTime` expects the value to be in valid `DateTime` format characters. Literal strings, values that are not placeholders for parts of the date, can be included in the date format by enclosing literal values in double quotes (within the format string):

```
PS> Get-Date -Format 'dddd" the "d" day of "MMM'
Sunday the 17 day of Jan
```

A similar outcome might be achieved using the format operator; for example:

```
'{0:dddd} the {0:dd} day of {0:MMM}' -f (Get-Date)
```

The format operator does not correctly interpret a single `d` to represent the day number.

The possible values that you can use in date format strings are listed with examples on Microsoft Docs:

https://docs.microsoft.com/dotnet/standard/base-types/standard-date-and-time-format-strings

https://docs.microsoft.com/dotnet/standard/base-types/custom-date-and-time-format-strings

When storing dates, it is a good practice to store them in an unambiguous format such as a universal date-time string. Consider the following:

```
(Get-Date).ToUniversalTime().ToString('u')
```

Alternatively, an ISO 8601 date can be created using this:

```
(Get-Date).ToUniversalTime().ToString('u').Replace(' ', 'T')
```

Many commands in PowerShell accept `DateTime` values for parameters. A string can be passed as an argument and PowerShell will attempt to coerce these values into `DateTime`.

DateTime parameters

While most commands deal with dates in a culture-specific format, care must be taken when passing dates as strings to parameters that cast to `DateTime` in cultures other than en-US.

Casting to `DateTime` does not account for a cultural bias. For example, in the UK, the format `dd/MM/yyyy` is used. Casting this format to `DateTime` switches the format to `MM/dd/yyyy` (as used in the US):

```
$string = "11/10/2000"     # 11th October 2000
[DateTime]$string          # 10th November 2000
```

If a function is created that accepts `DateTime` as a parameter, the result may not be as expected, depending on the local culture:

```
function Test-DateTime {
    param(
        [DateTime]$Date
    )
    $Date
}
Test-DateTime -Date "11/10/2000"
```

It is possible to work around this problem using the `Get-Date` command, to ensure that the culture-specific conversion is applied before `DateTime` is bound to the parameter (avoiding the attempt to coerce a string when binding a parameter):

```
Test-DateTime -Date (Get-Date "11/10/2000")
```

Commands such as `Test-Path` in PowerShell 7 accept a `DateTime` value for the `OlderThan` or `NewerThan` parameters. For example, the following command will only return `true` if a file exists in the specified path, and `LastWriteTime` is greater than one week ago:

```
Test-Path c:\temp\file.txt -NewerThan (Get-Date).AddDays(-7)
```

Event log search commands also make use of `DateTime`; for example, the following command finds the first 10 events from the system event log that were logged yesterday:

```
Get-WinEvent -MaxEvents 10 -FilterHashtable @{
    LogName = 'System'
    StartTime = (Get-Date).Date.AddDays(-1)
    EndTime   = (Get-Date).Date
}
```

Each of the dates used is found relative to the current date using the methods and properties explored in the previous section.

Comparing dates

`DateTime` objects may be compared using comparison operators:

```
$date1 = (Get-Date).AddDays(-20)
$date2 = (Get-Date).AddDays(1)
$date2 -gt $date1
```

Dates can be compared to a string; the value on the right-hand side will be converted into `DateTime`.

As with casting in parameters, care is required for date formats other than en-US. For example, in Europe, conversion might produce unexpected results if the `dd/mm/yyyy` format is used on the right-hand side:

```
PS> (Get-Date 13/01/2017) -gt '12/01/2017'
False
```

The comparison will succeed if the date is in en-US format:

```
PS> (Get-Date 13/01/2017) -gt '01/12/2017'
True
```

The result of the preceding operation depends on the current session culture. The date on the left, `13/01/2017`, is invalid if the culture is en-US and will cause `Get-Date` to show an error.

Dates are frequently used in PowerShell as parameter values, for comparisons, and to generate timestamp strings. The preceding methods and approaches allow dates to be easily manipulated.

Summary

The .NET methods and commands demonstrated in this chapter are an important part of any scripter or developer toolkit. The .NET methods used for strings, numbers, and date extend on or offer alternatives to the operators offered by PowerShell.

PowerShell includes tools to convert string or text formats to objects or collections of objects. For example, the CSV commands will convert a comma-delimited string into an object where columns can be worked with as properties of objects.

`ConvertFrom-StringData`, and the Windows PowerShell commands `Convert-String` and `ConvertFrom-String` offer advanced text parsing capabilities without the need to resort to regular expressions.

Base64 encoding was briefly introduced in this chapter, primarily focused on using the `-EncodedCommand` parameter of `powershell.exe` or `pwsh.exe`. Base64 encoding is commonly used to represent arrays of bytes.

Numeric operators, such as `+`, `-`, `%`, `*`, and `/` are a foundation for mathematical operations in PowerShell. The `System.Math` class explored in this chapter expands the range of operations to include rounding, raising a number to a power, square root, and many others.

Dates are an important data type; they are used by many things either as properties of objects, such as the `LastWriteTime` value of a file, or a parameter value, such as the `OlderThan` or `NewerThan` parameters of `Test-Path`.

Chapter 9, *Regular Expressions*, as the name suggests, explores using regular expressions in PowerShell.

9

Regular Expressions

You can use **regular expressions** (**regex**) to perform searches against a text. For the uninitiated, anything but a trivial regular expression can be a confusing mess. To make the topic more difficult, regular expressions differ slightly across different programming languages, platforms, and tools. Given that PowerShell is built on .NET, PowerShell uses .NET-style regular expressions. There are often several different ways to achieve a goal when using regular expressions.

This chapter covers the following topics:

- Regex basics
- Anchors
- Quantifiers
- Character classes
- Alternation
- Grouping
- Look-ahead and look-behind
- The .NET Regex type
- Regex options
- Examples of regular expressions

Regular expressions can be complex, but knowing a small set of basic characters provides a useful foundation to build upon.

Regex basics

A regular expression is a pattern consisting of literal characters, an exact match, and special characters that have a variety of different behaviors. For example, the dot character, `.`, matches any single character and not a literal dot.

A few basic characters can go a long way. Several of the most widely used characters and operators introduced in this section are summarized in *Table 9.1*:

Character	**Description**	**Example**
Any, except: `[\^$.|?*+()`	Literal character	`'a' -match 'a'`
`.`	Any single character (except carriage return, line feed, `\r`, and `\n`)	`'a' -match '.'`

`*`	The preceding character repeated zero or more times	`'abc' -match 'a*'` `'abc' -match '.*'`
`+`	The preceding character repeated one or more times	`'abc' -match 'a+'` `'abc' -match '.+'`
`\`	Escape a character's special meaning	`'*' -match '\*'` `'\' -match '\\'`
`?`	Optional character	`'abc' -match 'ab?c'` `'ac' -match 'ab?c'`

Table 9.1: Widely used regex characters

Literal characters

The best place to begin is with the simplest of expressions – expressions that contain no special characters. These expressions contain what are known as **literal characters**. A literal character can be anything except `[\^$.|?*+()`. You must escape special characters using `\` to avoid errors, as the following example shows:

```
'9*8'-match '\*'   # * is reserved
'1+5' -match '\+'   # + is reserved
```

Curly braces (`{}`) are considered literal in many contexts.

Regex is context sensitive

Curly braces become reserved characters if they enclose either a number, two numbers separated by a comma, or one number followed by a comma.

In the following two examples, `{` and `}` are literal characters:

```
'{string}' -match '{'
'{string}' -match '{string}'
```

In the preceding example, the curly braces take on a special meaning.

The following example looks to match the word `string` followed by `122` additional `g` characters (for a total of `123` `g` characters in the string):

```
'string{123}' -match 'string{123}'
```

Repetition using quantifiers is explored later in this chapter.

The following statement returns `True` and fills the `matches` automatic variable with what matched. The `matches` variable is a Hashtable; it's only updated when something successfully matches when using the `-match` operator:

```
PS> 'The first rule of regex club' -match 'regex'
True
PS> $matches
Name                           Value
----                           -----
0                              regex
```

If a `-match` fails, the `matches` variable continues to hold the last matching value:

```
PS> 'This match will fail' -match 'regex'
False
PS> $matches
Name                                                  Value
----                                                  -----
0                                                     regex
```

Any character (.)

The next step is to introduce the period, or dot (`.`). The dot matches any single character, except end-of-line characters. The dot character is therefore useful when capturing a string containing anything. The following statement returns `True` :

```
'abcdef' -match '......'
```

As the previous expression matches any six characters anywhere in a string, it will also return `True` when a longer string is provided. There are no implied boundaries on the length of a string, only on the number of characters matched:

```
'abcdefghijkl' -match '......'
```

Repetition with * and +

`+` and `*` are two of a set of characters known as **quantifiers**. Quantifiers define how many times the preceding character is repeated and are discussed in detail later in this chapter.

The `*` character can be used to repeat the preceding character zero or more times. Consider the following example:

```
'aaabc' -match 'a*'      # Returns true, matches 'aaa'
```

However, zero or more means the character in question does not have to be present at all:

```
'bcd' -match 'a*'          # Returns true, matches nothing
```

If a character must be present in a string, the `+` quantifier is more appropriate:

```
'aaabc' -match 'a+'        # Returns true, matches 'aaa'
'bcd' -match 'a+'          # Returns false
```

Combining `*` or `+` with `.` produces two very simple expressions: `.*` and `.+`. These expressions may be used as follows:

```
'Anything' -match '.*'    # 0 or more. Returns true
'' -match '.*'            # 0 or more. Returns true
'Anything' -match '.+'    # 1 or more. Returns true
```

Attempting to use either `*` or `+` as a match, without a preceding character, results in an error:

```
PS> '*' -match '*'
OperationStopped: Invalid pattern '*' at offset 1. Quantifier {x,y} following nothing.
```

The escape character (\)

`\` is an escape character, but it is perhaps more accurate to say that `\` changes the behavior of the character that follows. For example, finding a string that contains the normally reserved character `*` may be accomplished using `\`, as follows:

```
'1 * 3' -match '\*'
```

In the following example, `\` is used to escape the special meaning of `\`, making it a literal character:

```
'domain\user' -match 'domain\\user'
'domain\user' -match '.*\\.*'
```

The preceding expression matches zero or more of any character, followed by a literal `\`, and then zero or more of any character.

This technique may be used with `-replace` to change the domain prefix:

```
'domain\user' -replace 'domain\\', 'newdomain\'
```

Using `\` alone will result in either an invalid expression or an unwanted expression. For example, the following regular expression is valid, but it doesn't act as you might expect. The `.` character is treated as a literal value because it is escaped. The following `-match` will return `false`:

```
'domain\user' -match 'domain\.+'
```

The following string will be matched by the previous regular expression, as the string contains a literal `.` (repeated one or more times):

```
'domain.user' -match 'domain\.+'
```

The `-replace` operator allows you to manipulate parts of these strings as follows:

```
'Domain\User' -replace '.+\\'  # Everything up to and including \
```

Alternatively, it can `-replace` everything after a character:

```
'Domain\User' -replace '\\.+' # Everything including and after \
```

If an expression is being dynamically created, for instance, from a list of values or a value in a variable, the .NET `Regex` type can be used to escape values:

```
$values = @(
    'C:\Temp'
    'domain.net'
) | ForEach-Object { [Regex]::Escape($_) }
$regex = $values -join '|'
```

Once the above has been run, the result will be a single regex with any special characters escaped (making them literal). For example:

```
C:\\Temp|domain\.net
```

Any special characters in the strings are escaped, allowing the string to be incorporated into a regular expression as a literal value.

Optional characters

You can use the question mark character (`?`) to make the preceding character optional. For example, there might be a need to look for either the singular or plural form of a certain word:

```
'There are 23 sites in the domain' -match 'sites?'
```

The regular expression will match the optional `s` if it can; the `?` character is greedy. A greedy expression will match as many characters as it possibly can.

Non-printable characters

Regular expressions support searches for non-printable characters. The most common of these are shown in *Table 9.2*:

Description	Character
Tab	`\t`
Line feed	`\n`

Carriage return	`\r`

Table 9.2: Common non-printable characters

Debugging regular expressions

Regular expressions can quickly become complicated and difficult to understand. Modifying a complex regular expression is not a particularly simple undertaking.

While PowerShell indicates whether there is a syntax error in a regular expression, it cannot do more than that. For example, in the following expression, PowerShell announces that there is a syntax error:

```
PS> 'abc' -match '*'
OperationStopped: Invalid pattern '*' at offset 1. Quantifier {x,y} following nothing.
```

Fortunately, there are several websites that can help you to visualize a regular expression and lend to an understanding of how it works against a given string.

Debuggex is one such site. This service can pick apart regular expressions, showing how each element applies to an example. Debuggex can be found at `https://www.debuggex.com/`.

Debuggex uses Java regular expressions, so some of the examples used in this chapter may not be compatible.

Online engines that are .NET-specific, but do not include visualization, are as follows:

- `https://regextester.github.io/`
- `http://www.regexplanet.com/advanced/dotnet/index.html`

Finally, the website `http://www.regular-expressions.info` is an important learning resource that provides detailed descriptions, examples, and references.

Anchors

An anchor does not match a character; instead, it matches what comes before (or after) a character:

Description	**Character**	**Example**
Beginning of a string	`^`	`'aba' -match '^a'`
End of a string	`$`	`'cbc' -match 'c$'`

Word boundary	`\b`	`'Band and Land' -match '\band\b'`

Table 9.3: Commonly used anchors

Anchors are useful where a character, string, or word may appear elsewhere in a string and the position is critical.

For example, there might be a need to get values from the `PATH` environment variable that start with a specific drive letter. One approach to this problem is to use the start of a string anchor; in this case, retrieving everything that starts with the `C` drive:

```
$env:PATH -split ';' | Where-Object { $_ -match '^C' }
```

Given an array, the `-match` operator will return matching values, and `Where-Object` can be dropped:

```
$env:PATH -split ';' -match '^C'
```

Alternatively, there may be a need to get every path that is three or more directories deep from a given set:

```
$env:PATH -split ';' -match '\\.+\\.+\\.+$'
```

The word boundary anchor matches both before and after a word. It allows a pattern to look for a specific word, rather than a string of characters that may be a word or a part of a word.

For example, if the intent is to `-replace` the word `day` in the following string, then attempting this without the word boundary replaces too much:

```
PS> 'The first day is Monday' -replace 'day', 'night'
The first night is Monnight
PS> 'Monday is the first day' -replace 'day', 'night'
Monnight is the first night
```

Adding the word boundary avoids the problem of the expression matching the end of the world `Monday` without significantly increasing the complexity:

```
PS> 'The first day is Monday' -replace '\bday\b', 'night'
The first night is Monday
PS> 'Monday is the first day' -replace '\bday\b', 'night'
Monday is the first night
```

Repeating characters of parts of a regular expression is a common requirement. Repetition is controlled using quantifiers.

Quantifiers

A quantifier is used to define how many times a preceding character or group is repeated. Three examples of quantifiers have already been introduced: `*`, `+`, and `?`. The quantifiers are as follows:

Character	**Description**	**Example**

`*`	The preceding character repeated zero or more times	`'abc'-match 'a*'` `'abc'-match '.*'`
`+`	The preceding character repeated one or more times	`'abc'-match 'a+'` `'abc'-match '.+'`
`?`	Optional character	`'abc' -match 'ab?c'` `'ac' -match 'ab?c'`
`{exactly}`	A fixed number of characters	`'abbbc' -match 'ab{3}c'`
`{min,max}`	A number of characters within a range	`'abc' -match 'ab{1,3}c'` `'abbc' -match 'ab{1,3}c'` `'abbbc' -match 'ab{1,3}c'`
`{min,}`	Specifies a minimum number of characters	`'abbc' -match 'ab{2,}c'` `'abbbbbc' -match 'ab{2,}c'`

Table 9.4: Commonly used quantifiers

Each of `*`, `+`, and `?` can be described using curly brace notation:

- `*` is the same as `{0,}`
- `+` is the same as `{1,}`
- `?` is the same as `{0,1}`

It's extremely uncommon to find examples where the functionality of special characters is replaced with curly braces. It is equally uncommon to find examples where the quantifier `{1}` is used, as it adds unnecessary complexity to an expression.

Exploring the quantifiers

Each of these different quantifiers is greedy. A greedy quantifier grabs as much as it possibly can before allowing the regex engine to move on to the next character in the expression.

In the following example, the expression has been instructed to match everything it can, ending with a `\` character. As a result, it takes everything up to the last `\`, because the expression is greedy:

```
PS> 'C:\long\path\to\some\files' -match '.*\\'
True
PS> $matches[0]
C:\long\path\to\some\
```

The repetition operators can be made lazy by adding the `?` character. A lazy expression, by contrast, will get as little as it can before it ends:

```
PS> 'C:\long\path\to\some\files' -match '.*?\\'
True
PS> $matches[0]
C:\
```

The result of the match is everything up to the first `\` character.

A possible use of a lazy quantifier is parsing HTML. The following line describes a simple HTML table. The goal is to get the first table data (`td`) element:

```
<table><tr><td>Value1</td><td>Value2</td></tr></table>
```

Using a greedy quantifier will potentially take too much; the following expression will match all content between the first `<td>` and the last `</td>` (rather than just the first element):

```
PS> $html = '<table><tr><td>Value1</td><td>Value2</td></tr></table>'
PS> $html -match '<td>.+</td>'
True
PS> $matches[0]
<td>Value1</td><td>Value2</td>
```

Using a character class is one possible way to solve the problem of the expression matching too much. The character class (explored later in the chapter) is used to take all characters except `>`, which denotes the end of the next `</td>` tag:

```
PS> $html = '<table><tr><td>Value1</td><td>Value2</td></tr></table>'
PS> $html -match '<td>[^>]+</td>'
True
PS> $matches[0]
<td>Value1</td>
```

Another way to solve the problem of matching too much is to use a lazy quantifier:

```
PS> $html = '<table><tr><td>Value1</td><td>Value2</td></tr></table>'
PS> $html -match '<td>.+?</td>'
True
PS> $matches[0]
<td>Value1</td>
```

Quantifiers are a vital part of working with regular expressions. They are often used after character classes to describe repetition.

Character classes

A character class is used to match a single character to a set of possible characters. A character class is denoted using square brackets (`[ ]`).

For example, a character class may contain each of the vowels:

```
'get' -match 'g[aeiou]t'
'got' -match 'g[aeiou]'
```

Within a character class, the special or reserved characters are as follows:

- `-` : Used to define a range
- `\` : Escape character
- `^` : Negates the character class

Ranges

A hyphen is used to define a range of characters, for example, to capture any number of characters repeated one or more times in a set (using `+`):

```
'1st place' -match '[0-9]+'    # $matches[0] is "1"
'23rd place' -match '[0-9]+'   # $matches[0] is "23"
```

A range in a character class can be any range of ASCII characters, such as the following examples:

- a-z
- A-K
- 0-9
- 1-5
- !-9 (0-9 and the ASCII characters 33 to 47)

The following code returns `true` , as `"` is ASCII character `34` , and `#` is ASCII character `35` ; that is, they're within the specified `!-9` range:

```
PS> '"#' -match '[!-9]+'
True
PS> $matches[0]
"#
```

The range notation allows hexadecimal numbers within strings to be identified. A hexadecimal character can be identified by a character class containing `0-9` and `a-f` :

```
PS> 'The registry value is 0xAF9B7' -match '0x[0-9a-f]+'
True
PS> $matches[0]
0xAF9B7
```

If the comparison operator is case sensitive, the character class may also define `A-F`:

```
'The registry value is 0xAF9B7' -cmatch '0x[0-9a-fA-F]+'
```

Alternatively, you can use a range might to tentatively find an IP address in a string. The following example finds the IP address (or addresses) of the current Windows computer:

```
PS> (ipconfig) -match 'IPv4 Address.+: *[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+'
   IPv4 Address. . . . . . . . . . . : 172.16.255.30
```

The range used to find the IP address here is simple. It matches any string containing four numbers separated by a period. For example, the following version number matches this range:

```
'version 0.1.2.3234' -match '[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+'
```

This IP address-matching regular expression will be improved as the chapter progresses.

The hyphen is not a reserved character when it is put in a position where it does not describe a range. If it is the first character (with no start to the range), it will be treated as a literal. The following split operation demonstrates this:

```
PS> 'one-two_three,four' -split '[-_,]'
one
two
three
four
```

The same output is seen when `-` is placed at the end (that is, when there is no end to the range):

```
'one-two_three,four' -split '[_,-]'
```

Elsewhere in the class, the escape character may be used to remove the special meaning from the hyphen:

```
'one-two_three,four' -split '[_\-,]'
```

Negated character class

Within a character class, the caret (`^`) is used to negate the class. The character class `[aeiou]` matches vowels. Negating it with the caret, `[^aeiou]`, matches any character except a vowel (including spaces, punctuation, tabs, and everything else).

As with the hyphen, the caret is only effective if it is in the right position. In this case, it only negates the class if it is the first character. Elsewhere in the class, it is a literal character.

A negated character class is sometimes the fastest way to tackle a problem. If the list of expected characters is small, negating that list is a quick way to perform a match.

In the following example, the negated character class is used with the `-replace` operator to remove all non-alpha characters:

```
PS> 'Ba%by8 a12315t the1231 k#.,154eyboard' -replace '[^a-z ]'
Baby at the keyboard
```

Negated character classes are a useful technique to avoid attempting to list all the possible characters that should match. Character class subtraction adds even greater flexibility to this technique.

Character class subtraction

Character class subtraction is supported by .NET (and hence PowerShell). Character class subtraction is not commonly used at

all.

Inside a character class, a character class may be subtracted from another, reducing the size of the overall set. One of the best examples of this extends to the character class containing vowels. The following matches the first vowel in a string:

```
'The lazy cat sat on the mat' -match '[aeiou]'
```

To match the first consonant, one approach can be to list all the consonants:

```
'The lazy cat sat on the mat' -match '[b-df-hj-np-tv-z]'
```

Another approach to the problem is to take a larger character class, then subtract the vowels:

```
'The lazy cat sat on the mat' -match '[a-z-[aeiou]]'
```

Shorthand character classes

Several shorthand character classes are available. *Table 9.5* shows each of these:

Shorthand	Description	Character class
`\d`	Digit character	`[0-9]`
`\s`	White space (space, tab, carriage return, new line, and form feed)	`[ \t\r\n\f]`
`\w`	Word character	`[A-Za-z0-9_]`
`\D`	Not a digit character	`[^0-9]`
`\S`	Not a whitespace character	`[^ \t\r\n\f]`

`\W`	Not a word character	`[^A-Za-z0-9_]`

Table 9.5: Shorthand character classes

Each of these shorthand classes can be negated by capitalizing the letter. `[^0-9]` may be represented using `\D` and `\S` is for any character except white space, and `\W` for any character except a word character.

Unicode category class

The `\p` class may be used to match specific Unicode categories and is particularly useful for matching things like punctuation characters. `\p` cannot be used alone; a Unicode category must be defined in curly braces.

For example, the following expression matches any punctuation character (`{P}`):

```
PS> 'This, and that' -match '\p{P}'
True
PS> $matches
Name                           Value
----                           -----
0                              ,
```

The possible Unicode categories are documented on Microsoft Docs: https://docs.microsoft.com/dotnet/standard/base-types/character-classes-in-regular-expressions#SupportedUnicodeGeneralCategories.

A more specific match might be created by matching and replacing dash characters, including em- and en-dash variants:

```
'Get–Process' -replace '\p{Pd}', '-'
```

The preceding example may be hard to copy into the console. The following version recreates the dash character in the string using its character code, allowing the replacement to be demonstrated:

```
"Get$([char]8211)Process" -replace '\p{Pd}', '-'
```

The difference between the different dash types is hard to spot, yet en- and em-dashes can cause problems if used in place of a hyphen in PowerShell (outside of string values).

For instance, when a dash is used in a command name, PowerShell will not recognize the command, despite it appearing to be correct. The following example forces this problem to occur:

```
PS> & "Get$([char]8211)Process"
&: The term 'Get–Process' is not recognized as a name of a cmdlet, function, script file, or
executable program.
Check the spelling of the name, or if a path was included, verify that the path is correct and try
again.
```

Character classes are used to match a single character against a set. A character class only ever matches a single character. Alternation is used to match alternate sequences of longer strings.

Alternation

The alternation (OR) character in a regular expression is a pipe (`|`). This is used to combine several possible regular expressions.

A simple example is to match a list of words:

```
'one', 'two', 'three' -match 'one|three'
```

The alternation character has the lowest precedence; in the previous expression, every value is first tested against the expression to the left of the pipe and then against the expression to the right of the pipe.

The goal of the following expression is to extract strings that **only** contain the words `one` or `three`. Adding the start- and end-of-string anchors ensures that there is a boundary. However, because the left and right are treated as separate expressions, the result might not be as expected when using the following expression:

```
PS> 'one', 'one hundred', 'three', 'eighty three' -match '^one|three$'
one
one hundred
three
eighty three
```

The two expressions are evaluated as follows:

- Look for all strings that start with `one`
- Look for all strings that end with `three`

There are at least two possible solutions to this problem. The first is to add the start- and end-of-string characters to both expressions:

```
'one', 'one hundred', 'three', 'eighty three' -match '^one$|^three$'
```

Another possible solution is to use a group:

```
'one', 'one hundred', 'three', 'eighty three' -match '^(one|three)$'
```

Grouping is discussed in detail in the following section.

Grouping

A group in a regular expression serves several different possible purposes:

- To denote repetition (of more than a single character)
- To restrict alternation to a part of the regular expression
- To capture values

Repeating groups

Groups may be repeated using any of the quantifiers. The regular expression that tentatively identifies an IP address can be improved using a repeated group. The starting point for this expression is as follows:

```
[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+
```

In this expression, the `[0-9]+` term followed by a literal `.` character is repeated three times. Therefore, the expression can become as follows:

```
([0-9]+\.){3}[0-9]+
```

The expression itself is not specific, it will match much more than an IP address, but is also now more concise. This example will be explored further later in this chapter.

If `*` is used as the quantifier for the group, it becomes optional. If faced with a set of version numbers ranging in formats from `1` to `1.2.3.4`, a similar regular expression might be used:

```
[0-9]+(\.[0-9]+)*
```

The result of applying this to several different version strings is shown in the following code:

```
PS> 'v1', 'Ver 1.000.232.14', 'Version: 0.92', 'Version-7.92.1-alpha' |
>>     Where-Object { $_ -match '[0-9]+(\.[0-9]+)*' } |
>>     ForEach-Object { $matches[0] }
1
1.000.232.14
0.92
7.92.1
```

In the case of the last example, `-alpha` is ignored; if that were an interesting part of the version number, the expression would need to be modified to account for that.

Restricting alternation

Alternation is the lowest precedence operator. In a sense, it might be wise to consider it as describing an ordered list of regular expressions to test.

Placing an alternation statement in parentheses reduces the scope of the expression.

For example, it is possible to match a multi-line string using alternation as follows:

```
PS> $string = @'
First line
second line
third line
'@
PS> if ($string -match 'First(.|\r?\n)*line') { $matches[0] }
First line
second line
third line
```

In this example, as `.` does not match the end-of-line character, using alternation allows each character to be tested against a broader set. In this case, each character is tested to see whether it is any character, `\r\n` or `\n`.

A regular expression might be created to look for files with specific words, or parts of words, in the name:

```
Get-ChildItem -Recurse -File |
    Where-Object { $_.Name -match '(pwd|pass(word|wd)?).*\.(txt|doc)$' }
```

The expression that compares filenames looks for strings that contain `pwd`, `pass`, `password`, or `passwd`, followed by anything with the `.txt` or `.doc` extensions.

This expression will match any of the following (and more):

```
pwd.txt
server passwords.doc
passwd.txt
my pass.doc
private password list.txt
```

The preceding expression is rough and will match far more. It is not necessarily a great way to do more than a casual search for files.

Capturing values

The ability to capture values from a string is an incredibly useful feature of regular expressions.

When using the `-match` operator, groups that have been captured are loaded into the `matches` variable (Hashtable) in the order that they appear in the expression. Consider the following example:

```
PS> 'first second third' -match '(first) (second) (third)'
True
PS> $matches
Name Value
---- -----
3    third
2    second
1    first
0    first second third
```

The first key, `0`, is always the string that matched the entire expression. Numbered keys are added to the Hashtable for each of the groups in the order that they appear. This applies to nested groups as well, counting from the leftmost `(`:

```
PS> 'first second third' -match '(first) ((second) (third))'
True
PS> $matches
Name Value
---- -----
4    third
3    second
2    second third
1    first
0    first second third
```

When using the `-replace` operator, the `matches` variable is not filled, but the contents of individual groups are available as numbered values, which may be used in `Replace-With`:

```
PS>'first second third' -replace '(first) ((second) (third))', '$1, $4, $2'
first, third, second third
```

> **Use single quotes when tokens are included**
>
> As was mentioned in *Chapter 4*, *Operators*, single quotes should be used when using capture groups in `Replace-With`. Tokens in double quotes will expand as if they were PowerShell variables.

The capture groups used in the previous example are, by default, given a numeric *name* based on their position. A capture group can be given a specific name.

Named capture groups

Capture groups can be given names. The name must be unique within the regular expression.

The following syntax is used to name a group:

```
(?<GroupName>Expression)
```

This may be applied to the previous simple example as follows:

```
PS> 'first second third' -match '(?<One>first) (?<Two>second) (?<Three>third)'
True
PS> $matches
Name    Value
----    -----
One     first
Three   third
Two     second
0       first second third
```

In PowerShell, this adds a pleasant additional capability. If the goal is to tear apart text and turn it into an object, one approach is as follows:

```
if ('first second third' -match '(first) (second) (third)') {
    [PSCustomObject]@{
        One   = $matches[1]
        Two   = $matches[2]
        Three = $matches[3]
    }
}
```

This produces an object that contains the result of each (unnamed) `-match` group in a named property.

An alternative is to use named matches and create an object from the `matches` Hashtable. When using this approach, `$matches[0]` should be removed:

```
PS> if ('first second third' -match '(?<One>first) (?<Two>second) (?<Three>third)') {
    $matches.Remove(0)
    [PSCustomObject]$matches
}
One   Three Two
---   ----- ---
first third second
```

A possible disadvantage of this approach is that the output is not ordered, as it has been created from a Hashtable.

Named capture groups can also be used with the `-replace` operator. The name for the group can be used in `Replace-With` if the name is enclosed in curly braces:

```
PS> 'first second third' -replace
>>     '(?<One>first) (?<Two>second) (?<Three>third)',
>>     '${Three}, ${Two}, ${One}'
third, second, first
```

If a group is only present because of a quantifier or because of alternation, it can be excluded from `$matches` by making the group non-capturing.

Non-capturing groups

By default, every group is a capture group. A group can be marked as non-capturing by using `?:` before the expression. In the following example, the third group has been marked as a non-capturing group:

```
PS> 'first second third' -match '(?<One>first) (?<Two>second) (?:third)'
True
PS> $matches
Name   Value
----   -----
Two    second
One    first
0      first second third
```

This technique is useful when a group might need a quantifier, but when it is not desirable to include it in the `$matches` Hashtable.

In the following example, a new group is added around the last two groups. This group adds another:

```
PS> 'first second third' -match '(first) ((second) (third))'
True
PS> $matches
Name                           Value
----                           -----
4                              third
3                              second
2                              second third
1                              first
0                              first second third
```

Groups in matches are added in the order they are written in the regular expression. If it is not desirable to capture the content of the group, for instance, if the group is used to control repetition, it can be excluded using `?:`:

```
PS> 'first second third' -match '(first) (?:(second) (third))'
True
PS> $matches
Name                           Value
----                           -----
3                              third
2                              second
1                              first
0                              first second third
```

This technique may be useful when using `-replace`; it simplifies the list of values available even if an expression grows in complexity:

```
PS> 'first second third' -replace '(first) (?:(second) (third))',
>>     '$3, $2, $1'
third, second, first
```

Grouping, whether to repeat more extensive parts of an expression or to capture content, is an important part of working with regular expressions.

Look-ahead and look-behind

Look-ahead and look-behind may be used to describe what happens around an expression, without affecting what is captured or what is replaced.

You can use positive look-ahead and look-behind to match strings that begin or end with specific values. And, you can use negative look-ahead and look-behind to match strings that do not begin or end with specific values:

Expression	Description	Matches	Does not match
`one(?=, two)`	Positive look-ahead	one, two	one, three

`(?<=three, )four`	Positive look-behind	three, four	two, four
`one(?!, two)`	Negative look-ahead	one, three	one, two
`(?!=three, )four`	Negative look-behind	two, four	three, four

Table 9.6: Look-ahead and look-behind expressions

In each case in *Table 9.6*, the value in bold will be captured in `$matches[0]`. The value in the look-ahead or look-behind is not included in the match.

The two assertions can be combined to find a value, when that value is between two specific values, for example, a value between two HTML tags:

```
$string = @'
<table>
    <tr><td>1</td><td>One</td></tr>
    <tr><td>11</td><td>Eleven</td></tr>
    <tr><td>111</td><td>One-hundred and eleven</td></tr>
</table>
'@
$string -match '(?<=<td>\d<.+?><td>).+(?=</td>)'
```

The result of the match is `True` and the `$matches` variable is set as the following shows:

```
PS> $matches
Name                           Value
----                           -----
0                              One
```

The expression can be broken down into smaller parts.

The positive look-behind `(?<=<td>\d<.+?><td>)` matches a sequence of characters as follows:

- `<td>` – A literal value
- `\d` – A single-digit character
- `<` – A literal value
- `.+?` – One or more characters and as few characters as possible
- `>` – A literal value
- `<td>` – A literal value

This expression therefore matches the following value:

```
<td>1</td><td>
```

The main part of the regular expression is `.+`, which matches any string of one or more characters long. This string must come

immediately after the value matched by the positive look-behind.

Finally, the positive look-ahead `(?=</td>)` matches the literal value `<td>` only.

The expression will therefore match the string `One` as it is the only part of the string preceded by a single digit in the table data element.

For even more complex matching scenarios, the .NET `Regex` class can be extremely useful.

The .NET Regex type

Much of this chapter has been concerned with the `-match` operator. The `-match` operator is extremely useful but has a shortcoming: it returns a single contiguous string match from the value on the left-hand side of the operator; it cannot return more than one match except when using capture groups.

It is occasionally desirable to return all matches from a string. There are two ways to tackle this:

1. Use the `Select-String` command with the `AllMatches` parameter
2. Use the static `Matches` method of the `Regex` type

For example, the following string contains several key-value pairs:

```
$string = @'
Name: Ruth
Job title: Programmer
Language: C#
Level: Senior
'@
```

Each individual line can be broken down into a key and value pair using the following regular expression:

```
^(?<Key>[^:]+): (?<Value>.+)$
```

For example, the first line matches the expression. The value `Name` is placed in the group named `Key`. The value `Ruth` is placed in the group named `Value`.

If the start and end anchors are removed, and the expression is used with the `-match` operator, the first line only will be matched:

```
PS> $string -match '(?<Key>[^:]+): (?<Value>.+)'
True
PS> $matches
Name                           Value
----                           -----
Value                          Ruth
Key                            Name
0                              Name: Ruth
```

It is possible to split the line and use `Where-Object` to match each of the lines in turn to capture all the values. Those values might be added to an ordered dictionary before being converted to a custom object:

```
$properties = [Ordered]@{}
$string -split '\n' | Where-Object {
    $_ -match '^(?<Key>[^:]+): (?<Value>.+)$'
} | ForEach-Object {
    $properties[$matches['Key']] = $matches['Value']
}
[PSCustomObject]$properties
```

The result of this operation is:

```
Name   Job title    Language   Level
----   ---------    --------   -----
Ruth   Programmer   C#         Senior
```

An alternative approach is to use the static `Matches` method of the `Regex` type to extract a list of key and value pairs:

```
$string = @'
Name: Ruth
Job title: Programmer
Language: C#
Level: Senior
'@
$properties = [Ordered]@{}
[Regex]::Matches(
    $string,
    '^(?<Key>[^:]+): (?<Value>.+)$',
    'Multiline'
) | ForEach-Object {
    $properties[$_.Groups['Key'].Value] = $_.Groups['Value'].Value
}
[PSCustomObject]$properties
```

The result of either of the two preceding examples is a custom object created based on the content of the string:

```
Name    Job title    Language    Level
----    ---------    --------    -----
Ruth    Programmer   C#          Senior
```

When using the `Matches` static method, expressions are case-sensitive by default. The preceding example uses the `Multiline` option; if `IgnoreCase` were required too it can be added as shown here:

```
[Regex]::Matches(
    $string,
    '^(?<Key>[^:]+): (?<Value>.+)$',
    'Multiline, IgnoreCase'
)
```

This technique is especially useful for complex match and replace operations. The method returns a `MatchCollection` instance, which includes instances of the match as described in Microsoft Docs: https://docs.microsoft.com/dotnet/api/system.text.regularexpressions.match.

In addition to `Matches`, the `Regex` type provides the `Escape` method, used at the beginning of this chapter, and an `Unescape` method, which can be used to remove regex escape characters from a string.

The preceding `Multiline` and `IgnoreCase` options can be used with the `-match` operator if the options are written inside the regular expression.

Regex options

Regular expression options are used to control the behavior of certain characters in an expression.

Some aspects of the behavior of a regular expression can be modified by placing a regex option either at the beginning or around part of an expression.

In the following example, `.+` will only match the first line; the result of the whole match is in the `0` key in the `$matches` Hashtable:

```
PS> "First line`nSecond line" -match '.+'
True
PS> $matches[0]
First line
```

This is because dot (`.`) will not match line break characters by default. You can change this by setting a single-line option for the expression:

```
PS> "First line`nSecond line" -match '(?s).+'
True
PS> $matches[0]
First line
Second line
```

Table 9.7 shows various regex options and their effect:

Character	**Description**	**Effect**
`i`	Case-insensitive matching	Allows case-insensitive matching to be toggled on or off
`m`	Multi-line	`^` and `$` match the start and end of a line instead of the start and end of the string
`s`	Single-line	`.` also matches new-line characters
`n`	Ignore unnamed groups	Unnamed groups are not captured
`x`	Ignore unescaped white space	White space in the expression is ignored

Table 9.7: Regex options

You can use the case sensitivity option with both `-match` and `-cmatch`. With `-match`, the expression is case insensitive by default. It, or small parts of the expression, can be made case sensitive by doing the following:

```
PS> 'Name (first or full): Bob' -match '^[a-z]+.*(?-i)([A-Z].+)$'
True
PS> $matches[1]
Bob
```

When using `-cmatch`, the option might be enabled instead, for example:

```
'Name (first or full): Bob' -cmatch '^(?i)[a-z]+.*(?-i)([A-Z].+)$'
```

Options can be combined. In the following example, both single- and multi-line mode are enabled:

```
"First line`nSecond line" -match '(?sm)^\S+\sl.+$'
```

As `+` is greedy, the line break will be part of the matched string. If the lazy modifier is added to `+`, the match will stop at the end of the first line:

```
"First line`nSecond line" -match '(?sm)^\S+\sl.+?$'
```

You can enable options then disable them within an expression. This allows an option to apply to a small part of an expression if required:

```
PS> 'The Cat in the Hat' -cmatch '[A-Z][a-z]+ (?i)[a-z]+(?-i) [a-z ]+'
True
PS> $matches
Name                           Value
----                           -----
0                              The Cat in the
```

Options like these are perhaps more commonly used in languages other than PowerShell.

Perhaps one of the most useful options is the option to ignore unescaped white space. While it is not immediately obvious, it enables comments in regular expressions.

In a regex, a comment is denoted by a hash followed by a space at the start of a line. For example:

```
$regex = '(?x)# This is a comment
abc'
'abc' -match $regex
```

In the next section, you'll use this feature and cement the knowledge acquired from this chapter with a few examples.

Examples of regular expressions

The following examples walk you through creating regular expressions for several different formats.

MAC addresses

Media Access Control (**MAC**) is a unique identifier for network interfaces with 6-byte fields normally written in hexadecimal.

`ipconfig` with the `/all` switch parameter can show the MAC address of each interface. The address is written in hexadecimal, and each byte is separated by a hyphen, for example, `1a-2b-3c-4d-5f-6d`.

On Linux or Unix-based systems, the `ip address` command shows the hardware (MAC) address for interfaces. Each hexadecimal byte is separated by a `:`, such as `1a:2b:3c:4d:5f:6d`.

A regular expression can be created to simultaneously match these formats. The expression matches the output from the

`ipconfig /all` command on Windows and the `ip address` command on Linux. In both cases, the MAC address appears after a space at the end of a line.

Considering the preceding formats, the MAC address is made up of six groups of two hexadecimal characters.

To match a single hexadecimal character, the following character class may be used:

```
[0-9a-f]
```

The first byte is two characters, so a quantifier is added:

```
[0-9a-f]{2}
```

After the first byte, the string includes a hyphen or a colon:

```
[0-9a-f]{2}[-:]
```

This pattern of a byte followed by a hyphen or colon repeats five times:

```
([0-9a-f]{2}[-:]){5}
```

The final byte is not followed by a hyphen or colon; it is added to the end:

```
([0-9a-f]{2}[-:]){5}[0-9a-f]{2}
```

The hexadecimal string appears at the end of a line; the `$` anchor is added to describe this:

```
([0-9a-f]{2}[-:]){5}[0-9a-f]{2}$
```

Finally, the MAC address appears after a space:

```
\s([0-9a-f]{2}[-:]){5}[0-9a-f]{2}$
```

To exclude the space from the match, it is moved into a positive look-behind:

```
(?<=\s)([0-9a-f]{2}[-:]){5}[0-9a-f]{2}$
```

The regular expression is used in a `Where-Object` statement to find lines matching the MAC address. On Windows:

```
PS> $regex = '(?<=\s)([0-9a-f]{2}[-:]){5}[0-9a-f]{2}$'
PS> ipconfig /all  |
>>     Where-Object { $_ -match $regex } |
>>     ForEach-Object { $matches[0] }
12-34-56-78-9a-bc
```

Or Linux:

```
PS> $regex = '(?<=\s)([0-9a-f]{2}[-:]){5}[0-9a-f]{2}$'
PS> ip address |
>>     Where-Object { $_ -match $regex } |
>>     ForEach-Object { $matches[0] }
12:34:56:78:9a:bc
```

IP addresses

Validating an IPv4 address using a regular expression is not necessarily a trivial task.

The IP address consists of four octets; each octet can be a value between 0 and 255. When using a regular expression, the values are considered strings; therefore, the following strings must be considered:

- [0-9]: 0 to 9
- [1-9][0-9]: 1 to 9, then 0 to 9 (10 to 99)
- 1[0-9]{2}: 1, then 0 to 9, then 0 to 9 (100 to 199)
- 2[0-4][0-9]: 2, then 0 to 4, then 0 to 9 (200 to 249)
- 25[0-5]: 2, then 5, then 0 to 5 (250 to 255)

Each of these is an exclusive set, so alternation is used to merge all the previous small expressions into a single expression. This generates the following group, which matches a single octet (`0` to `255`):

```
([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])
```

For example, the following value will match successfully:

```
'243' -match '^([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])$'
```

Whereas the expression will not match `256` :

```
PS> '256' -match '^([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])$'
False
```

The IP address validation expression contains repetition now; it contains four octets with a period between each of them:

```
(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.){3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4]
[0-9]|25[0-5])
```

The IPAddress type might be better than regex

There are other, perhaps better, ways to test whether a string can be an IP address instead of using such a long regex.

If a string is a strong candidate for being an IP address, consider using the `TryParse` static method on the `IPAddress` type. It will handle both v4 and v6 addressing, as follows:

```
$ipAddress = [IPAddress]0 # Used as a placeholder
if ([IPAddress]::TryParse("::1", [ref]$ipAddress)) {
    $ipAddress
}
```

The regular expression is long and complicated. It can be broken down using the ignore expression white space option (which enables comments). The last octet is the same matching sequence as the first three; it does not have extra comments in this example:

```
$regex = '(?x)
    # First three octets
    ((
        # 0 to 9 OR
        [0-9]|
        # 10 to 99 OR
        [1-9][0-9]|
        # 100 to 199 OR
        1[0-9]{2}|
        # 200 to 249 OR
        2[0-4][0-9]|
        # 250 to 255
        25[0-5]
    # Trailing dot and repeat
    )\.){3}
    # Last octet
    ([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])'
'10.0.0.1' -match $regex
```

The preceding expression includes a lot of capture groups. These can be globally ignored since the values captured will not make a great deal of sense. You can add the *ignore unnamed groups* (`n`) option to the expression to ignore these, which saves adding `?:` to each of the groups individually:

```
$regex = '(?xn)
    # First three octets
    ((
        # 0 to 9 OR
        [0-9]|
        # 10 to 99 OR
        [1-9][0-9]|
        # 100 to 199 OR
        1[0-9]{2}|
        # 200 to 249 OR
        2[0-4][0-9]|
        # 250 to 255
        25[0-5]
    # Trailing dot and repeat
    )\.){3}
    # Last octet
    ([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])'
```

Using the expression with `match` will show it now only captures the one group:

```
PS> '10.0.0.1' -match $regex
True
PS> $matches
Name                           Value
----                           -----
0                              10.0.0.1
```

The netstat command

The `netstat` command produces a tab-delimited, fixed-width table. The following example converts the active connections that list active TCP connections, as well as listening TCP and UDP ports, into an object.

A snippet of the output that the example is intended to parse is shown in the following code:

```
PS> netstat -ano
Active Connections
  Proto  Local Address     Foreign Address    State        PID
  TCP    0.0.0.0:135       0.0.0.0:0          LISTENING    124
  TCP    0.0.0.0:445       0.0.0.0:0          LISTENING    4
  TCP    0.0.0.0:5357      0.0.0.0:0          LISTENING    4
```

When handling text such as this, a pattern based on white space (or not white space) can be used:

```
^\s*\S+\s+\S+
```

The expression breaks down as follows:

- `^` – The start of the string
- `\s*` – Zero or more white space characters; any padding to the left of the table
- `\S+` – One or more characters that are not white space; represents the value in the column, such as TCP
- `\s+` – One or more white space characters; the white space between columns

For each column, the following expression with a named group is created:

```
(?<ColumnName>\S+)\s+
```

The trailing `\s+` is omitted for the last column (`PID`).

The expression is long and benefits from the (`?x`) option, which ignores white space in the pattern and enables comments:

```
(?x)
# The left hand side of the table
^\s*
# Protocol column (and white space after)
(?<Protocol>\S+)\s+
# Local address column (and white space after)
(?<LocalAddress>\S+)\s+
# Foreign address column (and white space after)
(?<ForeignAddress>\S+)\s+
# State column (and white space after)
(?<State>\S+)\s+
# PID column and the end of string
(?<PID>\d+)$
```

The expression is long, but incredibly repetitive. The repetition is desirable in this case, where each column value is pushed into a differently named group.

The expression can be applied using `Where-Object` :

```
$regex = '(?x)
    # The left hand side of the table
    ^\s*
    # Protocol column (and white space after)
    (?<Protocol>\S+)\s+
    # Local address column (and white space after)
    (?<LocalAddress>\S+)\s+
    # Foreign address column (and white space after)
    (?<ForeignAddress>\S+)\s+
    # State column (and white space after)
    (?<State>\S+)\s+
    # PID column and the end of string
    (?<PID>\d+)$'
netstat -ano | Where-Object { $_ -match $regex } | ForEach-Object {
    $matches.Remove(0)
    [PSCustomObject]$matches
}
```

The output from this command will be missing information about UDP ports. The regular expression makes having a value in the state column mandatory. Marking this group as optional will add UDP connection information to the output:

```
(State>\S+)?
```

Inserting it back into the regular expression is achieved as follows:

```
$regex = '(?x)
    # The left hand side of the table
    ^\s*
    # Protocol column (and white space after)
    (?<Protocol>\S+)\s+
    # Local address column (and white space after)
    (?<LocalAddress>\S+)\s+
    # Foreign address column (and white space after)
    (?<ForeignAddress>\S+)\s+
    # State column (and white space after)
    (?<State>\S+)?\s+
    # PID column and the end of string
    (?<PID>\d+)$'
netstat -ano | Where-Object { $_ -match $regex } | ForEach-Object {
    $matches.Remove(0)
    [PSCustomObject]$matches
}
```

Finally, if it is desirable to return the fields in the same order as `netstat` does, `Select-Object` may be used:

```
$regex = '(?x)
    # The left hand side of the table
    ^\s*
    # Protocol column (and white space after)
    (?<Protocol>\S+)\s+
    # Local address column (and white space after)
    (?<LocalAddress>\S+)\s+
    # Foreign address column (and white space after)
    (?<ForeignAddress>\S+)\s+
    # State column (and white space after)
    (?<State>\S+)?\s+
    # PID column and the end of string
    (?<PID>\d+)$'
netstat -ano | Where-Object { $_ -match $regex } | ForEach-Object {
    $matches.Remove(0)
    [PSCustomObject]$matches
} | Select-Object Protocol, LocalAddress, ForeignAddress, State, PID |
    Format-Table
```

The outcome is a collection of custom objects that contain the values from `netstat`. The following shows a snippet of the output:

```
Protocol LocalAddress ForeignAddress State     PID
-------- ------------ -------------- -----     ---
TCP      0.0.0.0:135  0.0.0.0:0      LISTENING 124
TCP      0.0.0.0:445  0.0.0.0:0      LISTENING 4
TCP      0.0.0.0:5357 0.0.0.0:0      LISTENING 4
```

Formatting certificates

It is occasionally necessary to create certificates in specific formats to appease other systems.

A certificate may be exported as a Base64 string as the following code shows. The `InsertLineBreaks` option splits the string every 76 characters:

```
$certificate = Get-ChildItem Cert:\LocalMachine\Root |
    Select-Object -First 1
[Convert]::ToBase64String(
    $certificate.Export('Cert'),
    [System.Base64FormattingOptions]::InsertLineBreaks
)
```

If a different width is required, you can use a regular expression to tweak the format of the certificate, in this case, splitting at 64 characters wide:

```
$certificate = Get-ChildItem Cert:\LocalMachine\Root |
    Select-Object -First 1
@(
    '-----BEGIN CERTIFICATE-----'
    [Convert]::ToBase64String(
        $certificate.Export('Cert')
    ) -split '(?<=\G.{64})'
    '-----END CERTIFICATE-----'
) | Out-String
```

The previous example combines a positive look-behind with the `\G` anchor. The `\G` anchor matches from the end of the last match.

Summary

This chapter looked at regular expressions and their use in PowerShell.

The *Regex basics* section introduced several heavily used characters. We used anchors to show how the start and end of a string or word boundary may be used to restrict the scope of an expression.

Character classes were introduced as a powerful form of alternation, providing a range of options for matching a single character. We demonstrated alternation using different sets of expressions.

We looked at repetition using `"*"`, `+`, `?`, and curly braces (`{}`) and discussed the notion of greedy and lazy expressions.

Grouping was used as a means of limiting the scope of alternation to repeat larger expressions or to capture strings.

Finally, we went through several examples, bringing together the areas covered in this chapter to solve specific problems.

In *Chapter 10, Files, Folders, and the Registry*, we will discuss working with files, folders, and the registry.

10

Files, Folders, and the Registry

The filesystem and the registry are two among several providers available in PowerShell. A provider represents a data store in a hierarchy of container and leaf objects.

In the context of the filesystem, a container is a folder or directory. A leaf object is a file.

A provider is therefore a way to access arbitrary data that has been arranged to somewhat represent a filesystem.

This chapter covers the following topics:

- Working with providers
- Items
- Item properties and attributes
- Windows permissions
- Transactions
- File catalog commands

The commands used to work with data within a provider, such as a filesystem, are common to all providers.

Working with providers

Each provider shares a common set of commands, such as `Set-Location`, `Get-Item`, and `New-Item`.

The full list of commands that you can use when interacting with a provider can be seen by running the following snippet:

```
$params = @{
    Name = @(
        '*-Item*'
        '*-ChildItem'
        '*-Content'
        '*-Acl'
        '*-Location'
    )
    Module = @(
        'Microsoft.PowerShell.Management'
        'Microsoft.PowerShell.Security'
    )
}
Get-Command @params
```

Each group of commands (such as the `*-Content` commands) is used when a provider supports a certain behavior. This is indicated by the .NET type it inherits, and the interfaces it implements.

For example, a provider that supports navigation allows the use of `Get-Item`, `Get-ChildItem`, `Set-Location`, and so on. This is possible because the `FileSystemProvider` inherits from a `NavigationCmdletProvider` type:

```
PS> (Get-PSProvider FileSystem).ImplementingType.BaseType
IsPublic IsSerial Name                      BaseType
-------- -------- ----                      --------
True     False    NavigationCmdletProvider  System.Management.Automatio...
```

A provider that has content may allow the use of `Get-Content`, `Set-Content`, and so on. This is indicated by a provider supporting the `IContentCmdletProvider` interface:

```
PS> (Get-PSProvider FileSystem).ImplementingType.ImplementedInterfaces
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     False    IResourceSupplier
True     False    IContentCmdletProvider
True     False    IPropertyCmdletProvider
True     False    ISecurityDescriptorCmdletProvider
True     False    ICmdletProviderSupportsHelp
```

The preceding output also shows that the `FileSystem` provider implements the `ISecurityDescriptorCmdletProvider` interface. This indicates support for `Get-Acl` and `Set-Acl`.

In addition to the commands each provider supports, a provider can add dynamic parameters to a command. For example, in the certificate provider (the `cert:` drive), the `Get-ChildItem` command gains parameters for `CodeSigningCert`, `DocumentEncryptionCert`, `SSLServerAuthentication`, and so on.

These dynamic parameters are described in the help files for each provider. The help files are not currently available in PowerShell 7; you must use Windows PowerShell to view the content. For example:

```
Get-Help certificate
Get-Help registry
```

The name of the provider is taken from `Get-PSProvider`. Support for this help content is indicated by the `ICmdletProviderSupportsHelp` interface as shown for the preceding `FileSystem` provider.

The most common operation, especially with a provider like `FileSystem`, is navigating.

Navigating

`Set-Location`, which has the alias `cd`, is used to navigate around a provider's hierarchy; for example:

```
Set-Location \                # The root of the current drive
Set-Location Windows          # A child container named Windows
Set-Location ..               # Navigate up one level
Set-Location ..\..            # Navigate up two levels
Set-Location Cert:            # Change to a different drive
Set-Location HKLM:\Software   # Change to a specific child container under a drive
```

`Set-Location` may only be used to switch to a container object, such as a filesystem folder.

The **print working directory** (`$PWD`) variable shows the current location across all providers:

```
PS> $PWD
Path
----
HKLM:\Software\Microsoft\Windows\CurrentVersion
```

pwd and .NET

.NET classes and methods are oblivious to PowerShell's current directory. When the following command is executed, the file will be created in the `Start in` path (if a shortcut started PowerShell):

```
[System.IO.File]::WriteAllLines('file.txt', 'Some content')
```

.NET constructors and methods are an ideal place to use the `pwd` variable:

```
[System.IO.File]::WriteAllLines("$pwd\file.txt", 'Some content')
```

The `Set-Location` command changes the working directory to a new location. PowerShell also offers two commands that allow movement into and out of locations.

`Push-Location` will change the current location:

```
Push-Location c:\windows
```

`Pop-Location` returns to the previous location:

```
Pop-Location
```

In PowerShell 7, `Push-Location` and `Pop-Location` include a `StackName` parameter, which allows movement across several different lists of locations.

PowerShell 7 simplifies and enhances this by adding two special arguments to `Set-Location`. PowerShell tracks the locations visited with `Set-Location`. Using the `-` symbol for the path will move backward in the list of visited locations (somewhat equivalent to `Pop-Location`):

```
Set-Location c:\windows
# Return to the previous location
Set-Location -Path -
```

Using the `+` symbol for the path will move forward in the list of visited paths. Running the following example assumes `-` has been used beforehand, otherwise there will not be a location to move forward to:

```
Set-Location -Path +
```

These two values are synonymous with the back and forward buttons available in any web browser.

Getting items

The `Get-Item` command is used to get an instance of an object represented by a path:

```
Get-Item -Path \                           # The root container
Get-Item -Path .                           # The current container
Get-Item -Path ..                          # The parent container
Get-Item -Path C:\Windows\System32\cmd.exe # A leaf item
Get-Item -Path Cert:\LocalMachine\Root     # A container item
```

If a wildcard is used with the `Get-Item` command and the `Path` parameter, all matching items will be returned.

When working with the filesystem, a container is a directory (or folder) and a leaf is a file. In the registry, a key is a container and there are no leaves. In a certificate provider, a store or store location is a container, and a certificate is a leaf.

The `Get-ChildItem` command, which has the `dir`, `ls`, and `gci` aliases, is used to list the children of the current item.

Neither `Get-ChildItem` nor `Get-Item` will show hidden files and folders by default. The following error will be returned for a hidden item:

```
PS> Get-Item $env:USERPROFILE\AppData
Get-Item: Could not find item C:\Users\Chris\AppData.
```

The `Force` parameter may be added to access hidden items:

```
PS> Get-Item $env:USERPROFILE\AppData -Force
    Directory: C:\Users\Someone
Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d--h--       23/09/2016     18:22                AppData
```

Drives

PowerShell will automatically create a drive for any disk with a drive letter, any existing shared drive, the `HKEY_LOCAL_MACHINE` and `HKEY_CURRENT_USER` registry hives, the certificate store, and so on.

Additional drives may be added using `New-PSDrive`; for example, a network drive can be created:

```
New-PSDrive -Name X -PSProvider FileSystem -Root \\Server\Share
New-PSDrive -Name HKCR -PSProvider Registry -Root HKEY_CLASSES_ROOT
```

Existing drives may be removed using `Remove-PSDrive`. PowerShell allows filesystem drives to be removed; however, this is not a destructive operation, and it only removes the reference to the drive from PowerShell.

The filesystem provider supports the use of credentials when creating a drive, allowing network shares to be mapped using specific credentials.

Creating providers

It is possible to write limited providers in PowerShell by using the `SHiPS` module, which is available on the PowerShell Gallery: https://github.com/PowerShell/SHiPS.

Searching GitHub will show some projects using this: https://github.com/search?q=shipsdirectory&type=Code&utf8=%E2%9C%93.

Creating providers is beyond the scope of this chapter as it requires extensive use of PowerShell classes.

Items

Support for each of the `*-Item` commands varies from one provider to another. The filesystem provider supports all the commands, while the `Registry` provider supports a smaller number.

Testing for existing items

The `Test-Path` command may be used to test for the existence of a specific item under a drive:

```
Test-Path HKLM:\Software\Publisher
```

`Test-Path` distinguishes between item types with the `PathType` parameter. The container and leaf terms are used across providers to broadly classify items.

The following commands test for items of differing types:

```
Test-Path C:\Windows -PathType Container
Test-Path C:\Windows\System32\cmd.exe -PathType Leaf
```

Conversely, testing a path that is a file (leaf) with the container `PathType` will return false:

```
Test-Path C:\Windows\System32\cmd.exe -PathType Container
```

The `Test-Path` command is often used in an `if` statement prior to creating a file or directory to determine if creating the object is necessary:

```
if (-not (Test-Path C:\Temp\NewDirectory -PathType Container)) {
    New-Item C:\Temp\NewDirectory -ItemType Directory
}
```

In the preceding example, if a `C:\Temp\NewDirectory` file exists, the `New-Item` command will fail. `PathType` validation is perhaps more useful when validating parameters in functions and scripts than as an existence check before creating an object.

Windows PowerShell, Get-Item, Test-Path, and pagefile.sys

Some files in Windows are locked, with the result that `Get-Item` and `Test-Path` in Windows PowerShell are unable to correctly return results. The `pagefile.sys` file is one of these.

`Get-Item` returns an error, indicating that the file does not exist, even when the `Force` parameter is used. `Test-Path` always returns `false`.

This may be a bug in Windows PowerShell and is fixed in PowerShell 6 and above. To work around the problem, `Get-ChildItem` is able to get the file:

```
Get-ChildItem C:\ -Filter pagefile.sys -Force
```

To replace the functionality of `Test-Path`, the static method `Exists` may be used:

```
[System.IO.File]::Exists('c:\pagefile.sys')
```

Creating and deleting items

The `New-Item` command can create files, directories, keys, and so on depending on the provider:

```
New-Item $env:Temp\newfile.txt -ItemType File
New-Item $env:Temp\newdirectory -ItemType Directory
New-Item HKLM:\Software\NewKey -ItemType Key
```

When creating a file using `New-Item` in PowerShell, the file is empty (0 bytes).

In PowerShell 5, `New-Item` gained the ability to create symbolic links, junctions, and hard links:

- A symbolic link is a link to another file or directory. Creating a symbolic link requires administrator privileges (run as administrator).
- A hard link is a link to another file on the same drive.
- A junction is a link to another directory on any local drive. Creating a junction does not require administrative privileges.

You can create links as follows:

```
New-Item LinkName -ItemType SymbolicLink -Value \\Server\Share
New-Item LinkName.txt -ItemType HardLink -Value OriginalName.txt
New-Item LinkName -ItemType Junction -Value C:\Temp
```

Temporary files

If a script needs a file to temporarily store data, the `New-TemporaryFile` command may be used.

This command was introduced with PowerShell 5. Earlier versions of PowerShell may use the `Path.GetTempFileName` static method:

```
[System.IO.Path]::GetTempFileName()
```

Both commands create an empty file. The resulting file may be used with `Set-Content`, `Out-File`, or any commands that write data to a file.

The `Remove-Item` command may be used to remove an existing item under a provider; for example:

```
$file = New-TemporaryFile
Set-Content -Path $file -Value 'Temporary: 10'
Remove-Item $file
```

Providers such as a filesystem and registry are reasonably flexible about removing items. When removing a directory or key with children, the `recurse` parameter should be used.

The `certificate` provider restricts the use of `Remove-Item` to certificates; certificate stores cannot be added or removed.

Invoking items

`Invoke-Item` (which has an alias, `ii`) will open or execute an object using the default settings for that object:

```
# Open the current directory in explorer
Invoke-Item .
# Open test.ps1 in the default editor
Invoke-Item test.ps1
# Open cmd
Invoke-Item $env:windir\system32\cmd.exe
# Open the certificate store MMC for the current user
# Windows PowerShell only
Invoke-Item Cert:
```

The registry provider does not support `Invoke-Item`.

Items in each of the different providers may have one or more properties.

Item properties

The `Get-ItemProperty` and `Set-ItemProperty` commands allow individual properties to be modified.

What is meant by an item property varies from one provider to another. In the `FileSystem` provider, this can be an attribute of the file, such as `Hidden` or `System`. For the `Registry` provider, an item property is a value under a registry key.

Properties and the filesystem

When working with the filesystem provider, `Get-ItemProperty` and `Set-ItemProperty` are rarely needed. For example, `Set-ItemProperty` might be used to make a file read-only. The following example assumes that the `somefile.txt` file already exists:

```
Set-ItemProperty .\somefile.txt -Name IsReadOnly -Value $true
```

The same property may be set from a file object retrieved using `Get-Item` (or `Get-ChildItem`):

```
(Get-Item 'somefile.txt').IsReadOnly = $true
```

The `IsReadOnly` property affects the attributes of the file object, adding the `ReadOnly` attribute.

Adding and removing file attributes

The attributes of the file are used to describe if a file is hidden, or a system file, and so on.

The attributes property of a file object is a bit field presented as a number and given an easily understandable value by the

`System.IO.FileAttributes` enumeration.

> **Bit fields**
>
> A bit field is a means of exposing multiple settings that have two states (on or off binary states) using a single number.
>
> A byte, an 8-bit value, can therefore hold eight possible settings. A 32-bit integer, 4-bytes long, can hold 32 different settings.
>
> The following table, whose state is described by `4` bits, has four settings:
>
> **Name**: `Setting4 Setting3 Setting2 Setting1`
>
> **State**: `On Off On Off`
>
> **Binary**: `1 0 1 0`
>
> **Decimal**: `8 4 2 1`
>
> When settings `2` and `4` are toggled on, the value of the field is the conversion of `1010` to decimal. This value is the result of `8 -bor 2`, that is, `10`.

Several of the possible attributes are shown in *Table 10.1*:

Name	Compressed	Archive	System	Hidden	Read-only
Bit value	2048	32	4	2	1

Table 10.1: Attribute values

When a file is hidden and read-only, the value of the `Attributes` property is `3 (2 + 1)`. The value `3` can be cast to the `FileAttributes` type, which shows the names of the individual flags:

```
PS> [System.IO.FileAttributes]3
ReadOnly, Hidden
```

While the value is numeric, the use of the enumeration means words can be used to describe each property:

```
PS> [System.IO.FileAttributes]'ReadOnly, Hidden' -eq 3
True
```

This opens several possible ways to set attributes on a file.

Attributes may be replaced entirely:

```
(Get-Item 'somefile.txt').Attributes = 'ReadOnly, Hidden'
```

Attributes may be toggled:

```
$file = Get-Item 'somefile.txt'
$file.Attributes = $file.Attributes -bxor 'ReadOnly'
```

Attributes may be added:

```
$file = Get-Item 'somefile.txt'
$file.Attributes = $file.Attributes -bor 'ReadOnly'
```

The `+`, `-`, `+=`, and `-=` operators may be used, as this is a numeric operation. Addition or subtraction operations are not safe, as they do not account for existing flags. For example, if a file is already read-only and `+=` was used to attempt to make the file read-only, the result would be a hidden file:

```
PS> $file = Get-Item 'somefile.txt'
PS> $file.Attributes = 'ReadOnly'
PS> $file.Attributes += 'ReadOnly'
PS> $file.Attributes
Hidden
```

Finally, regardless of whether a flag is present, attributes may be written as a string:

```
$file = Get-Item 'somefile.txt'
$file.Attributes = "$($file.Attributes), ReadOnly"
```

This is a feasible approach because casting to the enumeration type will ignore any duplication:

```
PS> [System.IO.FileAttributes]'ReadOnly, Hidden, ReadOnly'
ReadOnly, Hidden
```

Registry values

`Get-ItemProperty` and `Set-ItemProperty` are most useful when manipulating registry values.

The following method may be used to get values from the registry:

```
Get-ItemProperty -Path HKCU:\Environment
Get-ItemProperty -Path HKCU:\Environment -Name Path
Get-ItemProperty -Path HKCU:\Environment -Name Path, Temp
```

Individual values may be written back to the registry under an existing key:

```
Set-ItemProperty -Path HKCU:\Environment -Name NewValue -Value 'New'
```

A registry value may be subsequently removed:

```
Remove-ItemProperty -Path HKCU:\Environment -Name NewValue
```

The `Set-ItemProperty` command does not directly allow the value type to be influenced. The command will do as much as it can to fit the value into the existing type. For a property with type `REG_SZ`, numbers will be converted to a string.

If a value does not already exist, a registry type will be created according to the value type:

- `Int32: REG_DWORD`
- `Int64: REG_QWORD`
- `String`: `REG_SZ`
- `String[]`: `REG_MULTI_SZ (must use "[String[]]@('value', 'value')")`
- `Byte[]`: `REG_BINARY`
- Any other type: `REG_SZ`

If a value of a specific type is required, the `New-ItemProperty` command should be used instead, for instance, if an expanding

string must be created:

```
$params = @{
    Path         = 'HKCU:\Environment'
    Name         = 'Expand'
    Value        = 'User: %USERNAME%'
    PropertyType = 'ExpandString'
}
New-ItemProperty @params
```

`New-ItemProperty` will throw an error if a property already exists. The `Force` parameter may be used to overwrite an existing value with the same name.

Windows permissions

Windows permissions, such as NTFS **Access Control Lists** (**ACLs**), are used to describe who or what can access a resource.

The filesystem and registry providers both support `Get-Acl` and `Set-Acl`, which allow the different ACLs to be modified.

Working with permissions in PowerShell involves a mixture of PowerShell commands and .NET objects and methods.

> **Alternatives to .NET classes**
>
> The `NtfsSecurity` module found in the PowerShell Gallery may be an easier alternative to the native methods discussed in this section.

While some values and classes differ between the different providers, many of the same concepts apply.

The following snippet creates a set of files and folders in `C:\Temp`. These files and folders are used in the examples that follow:

```
New-Item C:\Temp\ACL -ItemType Directory -Force
1..5 | ForEach-Object {
    New-Item C:\Temp\ACL\$_ -ItemType Directory -Force
    'content' | Out-File "C:\Temp\ACL\$_\$_.txt"
    New-Item C:\Temp\ACL\$_\$_ -ItemType Directory -Force
    'content' | Out-File "C:\Temp\ACL\$_\$_\$_.txt"
}
```

The `Get-Acl` command is used to retrieve an existing ACL for an object. `Set-Acl` is used to apply an updated ACL to an object.

If `Get-Acl` is used against a directory, the ACL type is `DirectorySecurity`; for a file, the ACL type is `FileSecurity`, and for a registry key, the ACL type is `RegistrySecurity`.

Access and audit

Access lists come with two different types of access controls.

The **Discretionary Access Control List** (**DACL**) is used to grant (or deny) access to a resource. The DACL is referred to as access in PowerShell.

The **System Access Control List** (**SACL**) is used to define which activities should be audited. The SACL is referred to as audit in PowerShell.

Reading and setting the audit ACL requires administrator privileges (run as administrator). `Get-Acl` will only attempt to read the audit ACL if it is explicitly requested. The `-Audit` switch parameter is used to request the list:

```
Get-Acl C:\Temp\ACL\1 -Audit | Format-List
```

As none of the folders created have audit ACLs at this time, the `-Audit` property will be blank.

Access Control Entries (**ACEs**) may be inherited from a parent container or may be explicitly defined. Inheritance is controlled by rule protection.

Rule protection

ACLs, by default, inherit rules (ACEs) from parent container objects. Access rule protection blocks the propagation of rules from a parent object.

Rule protection can be enabled for the access ACL using the `SetAccessRuleProtection` method or for the audit ACL using the `SetAuditRuleProtection` method on the ACL.

Setting rule protection has the same effect as disabling inheritance in the GUI.

Each of the methods expects two arguments. The first argument, `isProtected`, dictates whether the list should be protected. The second argument, named `preserveInheritance`, dictates what should be done with existing inherited entries. Inherited entries can either be copied or discarded.

In the following example, access rule protection is enabled (inheritance is disabled) and the previously inherited rules are copied into the ACL:

```
$acl = Get-Acl C:\Temp\ACL\2
$acl.SetAccessRuleProtection($true, $true)
Set-Acl C:\Temp\ACL\2 -AclObject $acl
```

Copied rules will only appear on the ACL (as explicit rules) after `Set-Acl` has been run.

If access rule protection is subsequently re-enabled, copied rules are not removed. The resulting ACL will contain both inherited and explicit versions of each of the rules. Inheritance can be re-enabled as follows:

```
$acl = Get-Acl C:\Temp\ACL\2
$acl.SetAccessRuleProtection($false, $false)
Set-Acl C:\Temp\ACL\2 -AclObject $acl
```

The ACL will have doubled in length:

```
PS> Get-Acl C:\Temp\ACL\2 |
>>     Select-Object -ExpandProperty Access |
>>     Select-Object FileSystemRights, IdentityReference, IsInherited
FileSystemRights             IdentityReference                 IsInherited
----------------             -----------------                 -----------
-536805376                   NT AUTHORITY\Authenticated Users  False
Modify, Synchronize          NT AUTHORITY\Authenticated Users  False
FullControl                  NT AUTHORITY\SYSTEM               False
268435456                    NT AUTHORITY\SYSTEM               False
268435456                    BUILTIN\Administrators            False
FullControl                  BUILTIN\Administrators            False
ReadAndExecute, Synchronize  BUILTIN\Users                     False
FullControl                  BUILTIN\Administrators            True
268435456                    BUILTIN\Administrators            True
FullControl                  NT AUTHORITY\SYSTEM               True
268435456                    NT AUTHORITY\SYSTEM               True
ReadAndExecute, Synchronize  BUILTIN\Users                     True
Modify, Synchronize          NT AUTHORITY\Authenticated Users  True
-536805376                   NT AUTHORITY\Authenticated Users  True
```

Discarding access rules will result in an empty ACL:

```
$acl = Get-Acl C:\Temp\ACL\3
$acl.SetAccessRuleProtection($true, $false)
Set-Acl C:\Temp\ACL\3 -AclObject $acl
```

Once this operation completes, any attempt to access the directory will result in access being denied:

```
PS> Get-ChildItem C:\Temp\ACL\3
Get-ChildItem: Access to the path 'C:\Temp\ACL\3' is denied.
```

You can restore access to the folder provided the current user has the `SeSecurityPrivilege` privilege. The privilege is granted when a process is run as administrator. Re-enabling inheritance is the simplest way to restore access:

```
$acl = Get-Acl C:\Temp\ACL\3
$acl.SetAccessRuleProtection($false, $false)
Set-Acl C:\Temp\ACL\3 -AclObject $acl
```

In the previous example, the second argument for `SetAccessRuleProtection`, `preserveInheritance`, is set to `false`. This value has no impact; it only dictates behavior when access rule protection is enabled.

Inheritance and propagation flags

Inheritance and propagation flags dictate how individual ACEs are pushed down to child objects.

Inheritance flags are described by the `System.Security.AccessControl.InheritanceFlags` enumeration. The possible values are as follows:

- `None`: Objects will not inherit this ACE
- `ContainerInherit`: Only container objects (such as directories) will inherit this entry
- `ObjectInherit`: Only leaf objects (such as files) will inherit this entry

Propagation flags are described by the `System.Security.AccessControl.PropagationFlags` enumeration. The possible values are as follows:

- `None`: Propagation of inheritance is not changed
- `NoPropagateInherit`: Do not propagate inheritance flags
- `InheritOnly`: This entry does not apply to this object, only children

These two flag fields are used to build the `Applies to` option shown in the GUI when setting security on a folder. *Table 10.2* shows how each option is created:

Option	**Flags**

This folder only	• Inheritance: `None` • Propagation: `None`
This folder, subfolders, and files	• Inheritance: `ContainerInherit`, `ObjectInherit` • Propagation: `None`
This folder and subfolders	• Inheritance: `ContainerInherit` • Propagation: `None`
This folder and files	• Inheritance: `ObjectInherit` • Propagation: `None`

Subfolders only	• Inheritance: `ContainerInherit` • Propagation: `InheritOnly`
Files only	• Inheritance: `ObjectInherit` • Propagation: `InheritOnly`

Table 10.2: GUI names and flag values

The `NoPropagateInherit` propagation flag comes into play when the "Only apply these permissions to objects and/or containers within this container" tick box is used. The tick box is available regardless of the object type and the flag is only applicable on container objects.

Removing ACEs

Individual rules may be removed from an ACL using several different methods:

- `RemoveAccessRule` : Matches `IdentityReference` and `AccessMask`
- `RemoveAccessRuleAll` : Matches `IdentityReference`
- `RemoveAccessRuleSpecific` : Exact match

Access mask is a generic term used to refer to specific rights granted (filesystem rights for a file or directory and registry rights for a registry key).

To demonstrate rule removal, explicit entries might be added to the ACL. Enabling and then disabling access rule protection will add new rules: the original inherited set and an explicitly set copy of the same rules.

To enable access rule protection and copy inherited rules, do the following:

```
$acl = Get-Acl C:\Temp\ACL\3
$acl.SetAccessRuleProtection($true, $true)
Set-Acl C:\Temp\ACL\3 -AclObject $acl
```

When disabling protection, once committed, the inherited rules will appear alongside the copied rules:

```
$acl = Get-Acl C:\Temp\ACL\3
$acl.SetAccessRuleProtection($false, $true)
Set-Acl C:\Temp\ACL\3 -AclObject $acl
```

You can view the rules in the ACL:

```
PS> $acl = Get-Acl C:\Temp\ACL\3
PS> $acl.Access |
>>     Select-Object IdentityReference, FileSystemRights, IsInherited
IdentityReference                          FileSystemRights IsInherited
-----------------                          ---------------- -----------
NT AUTHORITY\Authenticated Users                 -536805376       False
NT AUTHORITY\Authenticated Users        Modify, Synchronize       False
NT AUTHORITY\SYSTEM                             FullControl       False
NT AUTHORITY\SYSTEM                               268435456       False
BUILTIN\Administrators                            268435456       False
BUILTIN\Administrators                          FullControl       False
BUILTIN\Users                   ReadAndExecute, Synchronize       False
BUILTIN\Administrators                          FullControl        True
BUILTIN\Administrators                            268435456        True
NT AUTHORITY\SYSTEM                             FullControl        True
NT AUTHORITY\SYSTEM                               268435456        True
BUILTIN\Users                   ReadAndExecute, Synchronize        True
NT AUTHORITY\Authenticated Users        Modify, Synchronize        True
NT AUTHORITY\Authenticated Users                 -536805376        True
```

The following example finds each explicit rule and removes it from the ACL:

```
$acl = Get-Acl C:\Temp\ACL\3
$acl.Access | Where-Object IsInherited -eq $false | ForEach-Object {
    $acl.RemoveAccessRuleSpecific($_)
}
Set-Acl C:\Temp\ACL\3 -AclObject $acl
```

Copying lists and entries

You can copy access lists from one object to another; for example, a template ACL might have been prepared:

```
$acl = Get-Acl C:\Temp\ACL\4
$acl.SetAccessRuleProtection($true, $true)
$acl.Access |
    Where-Object IdentityReference -like '*\Authenticated Users' |
    ForEach-Object { $acl.RemoveAccessRule($_) }
Set-Acl C:\Temp\ACL\4 –AclObject $acl
```

You can apply this ACL to another object:

```
$acl = Get-Acl C:\Temp\ACL\4
Set-Acl C:\Temp\ACL\5 -AclObject $acl
```

If the ACL contains a mixture of inherited and explicit entries, the inherited entries will be discarded.

Access control rules may be copied in a similar manner:

```
# Get the ACE to copy
$ace = (Get-Acl C:\Temp\ACL\3).Access | Where-Object {
    $_.IdentityReference -like '*\Authenticated Users' -and
    $_.FileSystemRights -eq 'Modify, Synchronize' -and
    $_.IsInherited
}
# Get the target ACL
$acl = Get-Acl C:\Temp\ACL\5
# Add the entry
$acl.AddAccessRule($ace)
# Apply the change
Set-Acl C:\Temp\ACL\5 -AclObject $acl
```

ACEs may be created from scratch instead of being copied.

Adding ACEs

ACEs must be created before they can be added to an ACL.

Creating an ACE for the filesystem or the registry, and for access or audit purposes, uses a set of .NET classes:

- `System.Security.AccessControl.FileSystemAccessRule`
- `System.Security.AccessControl.FileSystemAuditRule`
- `System.Security.AccessControl.RegistryAccessRule`
- `System.Security.AccessControl.RegistryAuditRule`

There are several different ways to use these classes; this section focuses on the most common one.

Filesystem rights

The filesystem ACE uses the `System.Security.AccessControl.FileSystemRights` enumeration to describe the different rights that might be granted.

PowerShell can list each right name using the `GetNames` (or `GetValues`) static methods of the `Enum` type:

```
[System.Security.AccessControl.FileSystemRights].GetEnumNames()
```

PowerShell might be used to show the names, numeric values, and even the binary values associated with each. Several of these rights are composites, such as write, which summarizes `CreateFiles`, `AppendData`, `WriteExtendedAttributes`, and `WriteAttributes`:

```
[System.Security.AccessControl.FileSystemRights].GetEnumValues() |
    Select-Object -Unique |
    ForEach-Object {
        [PSCustomObject]@{
            Name   = $_
            Value  = [Int]$_
            Binary = [Convert]::ToString([Int32]$_, 2).PadLeft(32, '0')
        }
    }
```

Microsoft Docs is a better place to find a descriptive meaning of each of the different flags:

https://docs.microsoft.com/dotnet/api/system.security.accesscontrol.filesystemrights

`FileSystemRights` is a bit field and can therefore be treated in the same way as `FileAttributes` earlier in this chapter. The simplest way to present rights is in a comma-separated list. There are many possible combinations; the GUI shows a small number of these before heading into advanced options. *Table 10.3* shows the main options:

GUI option	**Filesystem rights**
Full control	`FullControl`
Modify	`Modify`, `Synchronize`
Read and execute	`ReadAndExecute`, `Synchronize`
List folder contents	`ReadAndExecute`, `Synchronize`
Read	`Read`, `Synchronize`
Write	`Write`, `Synchronize`

Table 10.3: GUI options

Table 10.3 shows that both *Read and execute* and *List folder contents* have the same value. This is because the access mask is the same; the difference is in the inheritance flags:

GUI option	**Inheritance flags**
Read and execute	`ContainerInherit`, `ObjectInherit`

List folder contents	`ContainerInherit`

Table 10.4: Inheritance flags

In all other cases, the inheritance flags are set to `ContainerInherit` and `ObjectInherit`. Propagation flags are set to `None` for all examples.

Using these, you can create a full control ACE using one of the constructors for `FileSystemAccessRule`:

```
$ace = [System.Security.AccessControl.FileSystemAccessRule]::new(
    'DOMAIN\User',                        # Identity reference
    'FullControl',                        # FileSystemRights
    'ContainerInherit, ObjectInherit',    # InheritanceFlags
    'None',                               # PropagationFlags
    'Allow'                               # ACE type (allow or deny)
)
```

If the ACE is changed to use a valid user, the ACE can be applied to the ACL:

```
$acl = Get-Acl C:\Temp\ACL\5
$acl.AddAccessRule($ace)
Set-Acl C:\Temp\ACL\5 -AclObject $acl
```

Registry rights

Creating ACEs for registry keys follows the same pattern as for filesystem rights. The rights are defined in the `System.Security.AccessControl.RegistryRights` enumeration.

PowerShell can list these rights, but the descriptions on MSDN are more useful: https://msdn.microsoft.com/en-us/library/system.security.accesscontrol.registryrights(v=vs.110).aspx.

A rule is created in the same way as a filesystem rule:

```
$ace = [System.Security.AccessControl.RegistryAccessRule]::new(
    'DOMAIN\User',                        # Identity reference
    'FullControl',                        # RegistryRights
    'ContainerInherit, ObjectInherit',    # InheritanceFlags
    'None',                               # PropagationFlags
    'Allow'                               # ACE type (allow or deny)
)
```

The rule can be applied to a key (in this case, a newly created key):

```
$key = New-Item HKCU:\TestKey -ItemType Key -Force
$acl = Get-Acl $key.PSPath
$acl.AddAccessRule($ace)
Set-Acl $key.PSPath -AclObject $acl
```

Each of the access rights, such as `FullControl` used above, is described by an enumeration – in the previous example, the `RegistryRights` enumeration.

Numeric values in the ACL

The `FileSystemRights` enumeration used in the previous examples does not quite cover all the possible values one might see when inspecting an ACL. In some cases, the rights will be shown as numeric values rather than names. The `-536805376` and `268435456` values were both included in some earlier examples. The missing values are part of the generic portion of the ACE

in Microsoft Docs:

https://docs.microsoft.com/windows/desktop/SecAuthZ/access-mask-format

This generic portion is not accounted for by the `FileSystemRights` enumeration. These generic values, in turn, represent summarized rights:

https://docs.microsoft.com/windows/desktop/FileIO/file-security-and-access-rights

Converting each of the values to binary goes a long way in showing their composition:

```
PS> foreach ($value in -536805376, 268435456) {
>>     '{0,-10}: {1}' -f @(
>>         $value
>>         [Convert]::ToString($value, 2).PadLeft(32, '0')
>>     )
>> }
-536805376: 11100000000000010000000000000000
268435456 : 00010000000000000000000000000000
```

This script uses a `GenericAccessRights` enumeration to show how these values may be deconstructed:

```
using namespace System.Security.AccessControl
# Define an enumeration which describes the generic access mask (only)
[Flags()]
enum GenericAccessRights {
    GenericRead    = 0x80000000
    GenericWrite   = 0x40000000
    GenericExecute = 0x20000000
    GenericAll     = 0x10000000
}
# For each value to convert
foreach ($value in -536805376, 268435456) {
    # For each enum that describes the values
    $accessRights = foreach ($enum in [GenericAccessRights], [FileSystemRights]) {
        # Find values from the enum where the value with the exact bit set.
        [Enum]::GetValues($enum) | Where-Object {
            ($value -band $_) -eq $_
        }
    }
    # Output the original value and the values from the enum (as a string)
    '{0} : {1}' -f $value, ($accessRights -join ', ')
}
```

The two values discussed are therefore the following:

- `-536805376` : `GenericExecute` , `GenericWrite` , `GenericRead` , and `Delete`
- `268435456` : `GenericAll`

The security descriptor describes the owner of the object. This owner can be changed.

Ownership

Ownership of a file or directory may be changed using the `SetOwner` method of the ACL object. Changing the ownership of a file requires administrative privileges.

The owner of the `C:\Temp\ACL\1` file is the current user:

```
PS> Get-Acl C:\Temp\ACL\1 | Select-Object Owner
Owner
-----
COMPUTER\Chris
```

The owner may be changed (in this case, to the `Administrator` account) using the `SetOwner` method:

```
$acl = Get-Acl C:\Temp\ACL\1
$acl.SetOwner([System.Security.Principal.NTAccount]'Administrator')
Set-Acl C:\Temp\ACL\1 -AclObject $acl
```

> **This is not taking ownership**
>
> Setting ownership when the current user already has full control is one thing. Specific privileges are required to take ownership without existing permissions: `SeRestorePrivilege`, `SeBackupPrivilege`, and `SeTakeOwnershipPrivilege`.

Managing security in PowerShell can be complicated without the use of modules dedicated to the task. However, an understanding of how to work with ACLs in this manner is important as many different systems utilize the same generalized security objects and not all will have friendly modules.

Transactions

A transaction allows a set of changes to be grouped together and committed at the same time. Transactions are only supported under Windows PowerShell.

The registry provider supports transactions in Windows, as shown in the following code:

```
PS> Get-PSProvider
Name            Capabilities                         Drives
----            ------------                         ------
Registry        ShouldProcess, Transactions          {HKLM, HKCU}
Alias           ShouldProcess                        {Alias}
Environment     ShouldProcess                        {Env}
FileSystem      Filter, ShouldProcess, Credentials   {B, C, D}
Function        ShouldProcess                        {Function}
Variable        ShouldProcess                        {Variable}
```

You can create a transaction as follows:

```
Start-Transaction
$path = 'HKCU:\TestTransaction'
New-Item $path -ItemType Key -UseTransaction
Set-ItemProperty $path -Name 'Name' -Value 'Transaction' -UseTransaction
Set-ItemProperty $path -Name 'Length' -Value 20 -UseTransaction
```

At this point, you can undo the transaction:

```
Undo-Transaction
```

Alternatively, the transaction may be committed:

```
Complete-Transaction
```

A list of the commands that support transactions may be viewed, although not all of these may be used with the registry

provider:

```
Get-Command -ParameterName UseTransaction
```

Transactions are an interesting feature, but they are only supported by the registry provider and were not available in the earliest versions of .NET Core. The latest versions of .NET Core (and .NET 5 and higher) do provide support, but the commands used above are not part of PowerShell 7 and may never return.

File catalog commands

The file catalog commands were added in Windows PowerShell 5.1. A file catalog is a reasonably lightweight form of **File Integrity Monitoring** (**FIM**). The file catalog generates and stores SHA1 hashes for each file within a folder structure and writes the result to a catalog file.

> **About hashing**
>
> Hashing is a one-way process; a hash is not an encryption or encoding. A hash algorithm converts data of any length to a fixed-length value. The length of the value depends on the hashing algorithm used.
>
> MD5 hashing is one of the more common algorithms; it produces a 128-bit hash that can be represented by a 32-character string.
>
> SHA1 is rapidly becoming the default; it produces a 160-bit hash that can be represented by a 40-character string.
>
> PowerShell has a `Get-FileHash` command that can be used to calculate the hash for a file.

As the catalog is the basis for determining integrity, it should be maintained in a secure location, away from the set of files being analyzed.

You can create a small set of directories and files to help demonstrate the commands in the proceeding subsections:

```
New-Item C:\Temp\FileCatalog -ItemType Directory -Force
1..5 | ForEach-Object {
    New-Item C:\Temp\FileCatalog\$_ -ItemType Directory -Force
    'content' | Out-File "C:\Temp\FileCatalog\$_\$_.txt"
    New-Item C:\Temp\FileCatalog\$_\$_ -ItemType Directory -Force
    'content' | Out-File "C:\Temp\FileCatalog\$_\$_\$_.txt"
}
```

You will modify these files to demonstrate how the `FileCatalog` commands show changes.

New-FileCatalog

The `New-FileCatalog` command is used to generate (or update) a catalog:

```
New-FileCatalog -Path <ToWatch> -CatalogFilePath <StateFile>
```

A hash can only be generated for files that are larger than 0 bytes. However, filenames are recorded irrespective of the size.

The following command creates a file catalog from the files and folders created when exploring permissions:

```
$params = @{
    Path            = 'C:\Temp\FileCatalog'
    CatalogFilePath = 'C:\Temp\Security\example.cat'
}
New-FileCatalog @params
```

The only output from this command is an object representing the catalog file. The `Security` folder used above will be created if it does not already exist.

If the `CatalogFilePath` had been a directory instead of a file, `New-FileCatalog` would have automatically created a file named `catalog.cat`.

Test-FileCatalog

The `Test-FileCatalog` command compares the content of the catalog file to the filesystem. Hashes are recalculated for each file.

If none of the content has changed, `Test-FileCatalog` will return `Valid`; this can be seen by running the following command:

```
$params = @{
    Path            = 'C:\Temp\FileCatalog'
    CatalogFilePath = 'C:\Temp\Security\example.cat'
}
Test-FileCatalog @params
```

If a file has been added, removed, or changed, the `Test-FileCatalog` command will return `ValidationFailed`:

```
Set-Content C:\Temp\FileCatalog\3\3.txt -Value 'New content'
$params = @{
    Path            = 'C:\Temp\FileCatalog'
    CatalogFilePath = 'C:\Temp\Security\example.cat'
}
Test-FileCatalog @params
```

At this point, the `Detailed` parameter can be used to see what has changed.

> **Is it faster without Detailed?**
>
> The `Detailed` parameter does not change the amount of work `Test-FileCatalog` must do. If the result is to be used, it might be better to use the `Detailed` parameter right away. This saves the CPU cycles and I/O operations required to list the content of a directory and generate the hashes a second time.

The command does not provide a summary of changes; instead, it returns all files and hashes from the catalog and all files and hashes from the path being tested:

```
PS> Set-Content C:\Temp\FileCatalog\3\3.txt -Value 'New content'
PS> $params = @{
>>     Path            = 'C:\Temp\FileCatalog'
>>     CatalogFilePath = 'C:\Temp\Security\example.cat'
>>     Detailed        = $true
>> }
PS> Test-FileCatalog @params
Status        : ValidationFailed
HashAlgorithm : SHA1
CatalogItems  : {[1\1.txt, 3AC201172677076A818A18EB1E8FEECF1A04722A...}
PathItems     : {[1\1.txt, 3AC201172677076A818A18EB1E8FEECF1A04722A...}
Signature     : System.Management.Automation.Signature
```

Before exploring the output, a few more changes should be made:

```
Set-Content C:\Temp\FileCatalog\3\3-1.txt -Value 'New file'
Remove-Item C:\Temp\FileCatalog\1\1.txt
```

You can use these values to find changes. First, assign the result of the command to a variable:

```
$params = @{
    Path            = 'C:\Temp\FileCatalog'
    CatalogFilePath = 'C:\Temp\Security\example.cat'
    Detailed        = $true
}
$result = Test-FileCatalog @params
```

The `PathItems` and `CatalogItems` properties are dictionary objects that contain the relative file path as a key, and the hash as the value. For example:

```
PS> $result.PathItems
Key       Value
---       -----
2\2.txt   3AC201172677076A818A18EB1E8FEECF1A04722A
3\3-1.txt 74A5D5EBDB364E61A6FB15090A2009937473312F
3\3.txt   0E2126C909E867CD180E140CA501CE52533C381F
4\4.txt   3AC201172677076A818A18EB1E8FEECF1A04722A
5\5.txt   3AC201172677076A818A18EB1E8FEECF1A04722A
1\1\1.txt 3AC201172677076A818A18EB1E8FEECF1A04722A
2\2\2.txt 3AC201172677076A818A18EB1E8FEECF1A04722A
3\3\3.txt 3AC201172677076A818A18EB1E8FEECF1A04722A
4\4\4.txt 3AC201172677076A818A18EB1E8FEECF1A04722A
5\5\5.txt 3AC201172677076A818A18EB1E8FEECF1A04722A
```

The `$result` variable may now be used to describe the changes. Files that have been added can be listed with the following code:

```
$result.PathItems.Keys | Where-Object {
    -not $result.CatalogItems.ContainsKey($_)
}
```

This will show that the file `3\3-1.txt` has been created. The hash of that file can be seen from `PathItems` as shown here:

```
PS> $result.PathItems['3\3-1.txt']
74A5D5EBDB364E61A6FB15090A2009937473312F
```

Files that have been removed are listed with the following:

```
$result.CatalogItems.Keys | Where-Object {
    -not $result.PathItems.ContainsKey($_)
}
```

If the file was removed above, then `1\1.txt` will show in the output. Since `PathItems` does not contain that file, the hash must be retrieved from `CatalogItems` if it is important:

```
PS> $result.CatalogItems['1\1.txt']
3AC201172677076A818A18EB1E8FEECF1A04722A
```

Files that have been added or modified are listed with the following code:

```
$result.PathItems.Keys | Where-Object {
    $result.CatalogItems[$_] -ne $result.PathItems[$_]
}
```

This command should show both `3\3-1.txt` and `3\3.txt`. To find changes only, the key must be present in both `CatalogItems` and `PathItems`:

```
$result.PathItems.Keys | Where-Object {
    $result.CatalogItems.ContainsKey($_) -and
    $result.CatalogItems[$_] -ne $result.PathItems[$_]
}
```

Running this command will show that the only changed file is `3\3.txt`.

As the file catalog only stores hashes, the command is unable to describe exactly what has changed about a file, only that something has.

Summary

This chapter looked at working with providers, focusing on filesystem and registry providers.

Providers use a common set of commands to access data arranged in a hierarchy. Providers may choose to add extra functionality with dynamic parameters for each of the commands, for example, by adding parameters to provide filtering.

Provider implementations can choose to support a variety of different operations, from reading and writing content to management ACLs.

In Windows PowerShell, the `Registry` provider supports transactions, allowing a sequence of changes to be prepared, then either committed or undone as applicable.

PowerShell 5 added commands to work with file catalogs. You can use a file catalog to see how a set of files is changing over time or to validate a copied folder.

Chapter 11, Windows Management Instrumentation, will explore how to work with WMI using the CIM commands built into Windows PowerShell and PowerShell Core.

11

Windows Management Instrumentation

Windows Management Instrumentation (**WMI**) was introduced as a downloadable component with Windows 95 and NT. Windows 2000 had WMI pre-installed, and it has since become a core part of the operating system.

You can use WMI to access a huge amount of information about the computer system. This includes printers, device drivers, user accounts, ODBC, and so on; there are hundreds of classes to explore.

In this chapter, we will be covering the following topics:

- Working with WMI
- CIM cmdlets
- The WMI Query Language
- WMI type accelerators
- Permissions

Working with WMI

The scope of WMI is vast, which makes it a fantastic resource for automating processes. WMI classes are not limited to the core operating system; it is not uncommon to find classes created after software or device drivers have been installed.

Given the scope of WMI, finding an appropriate class can be difficult. PowerShell itself is well equipped to explore the available classes.

WMI classes

PowerShell, as a shell for working with objects, presents WMI classes in a similar manner to .NET classes or any other object. There are several parallels between WMI classes and .NET classes.

A WMI class is used as the recipe to create an instance of a WMI object. The WMI class defines properties and methods. The WMI class `Win32_Process` is used to gather information about running processes in a similar manner to the `Get-Process` command.

The `Win32_Process` class has properties such as `ProcessId`, `Name`, and `CommandLine`. It has a `terminate` method that can be used to kill a process, as well as a `create static` method that can be used to spawn a new process.

WMI classes reside within a WMI namespace. The default namespace is `root\cimv2`; classes such as `Win32_OperatingSystem` and `Win32_LogicalDisk` reside in this namespace.

WMI commands

PowerShell has two different sets of commands dedicated to working with WMI.

`Get-WmiObject` is included with PowerShell 1.0 to 5.1 but is not present in PowerShell 6 and above.

The CIM cmdlets were introduced with PowerShell 3.0. They are compatible with the **Distributed Management Task Force** (**DMTF**) standard DSP0004. A move toward compliance with open standards is critical as the Microsoft world becomes more diverse.

WMI itself is a proprietary implementation of the CIM server, using the **Distributed Component Object Model** (**DCOM**) API to communicate between the client and server.

Standards compliance and differences in approach aside, there are solid, practical reasons to consider when choosing which one to use.

Some properties of CIM cmdlets are as follows:

- They are available in both Windows PowerShell and PowerShell Core.
- They handle date conversion natively.
- They have a flexible approach to networking. They use `WSMAN` for remote connections by default but can be configured to use DCOM over RPC.
- They can be used for all WMI operations.

Some properties of WMI cmdlets are as follows:

- They are only available in Windows PowerShell and are not in PowerShell Core
- They do not automatically convert dates
- They use DCOM over RPC exclusively
- They can be used for all WMI operations
- They have been superseded by the CIM cmdlets

CIM cmdlets

As mentioned in the previous section, the **Common Information Model** (**CIM**) cmdlets were introduced with PowerShell 3 and are the only commands available to access WMI in PowerShell 6 and above.

The CIM commands are as follows:

- `Get-CimAssociatedInstance`
- `Get-CimClass`
- `Get-CimInstance`
- `Get-CimSession`
- `Invoke-CimMethod`
- `New-CimInstance`
- `New-CimSession`
- `New-CimSessionOption`
- `Register-CimIndicationEvent`
- `Remove-CimInstance`
- `Remove-CimSession`
- `Set-CimInstance`

Each of these CIM cmdlets uses either the `ComputerName` or `CimSession` parameter to target the operation at another computer.

Getting instances

You can use the `Get-CimInstance` command to execute queries for instances of WMI objects, as the following code shows:

```
Get-CimInstance -ClassName Win32_OperatingSystem
Get-CimInstance -ClassName Win32_Service
Get-CimInstance -ClassName Win32_Share
```

Several different parameters are available when using `Get-CimInstance`. The command can be used with a filter, as follows:

```
Get-CimInstance Win32_Directory -Filter "Name='C:\\Windows'"
Get-CimInstance Win32_Service -Filter "State='Running'"
```

Each command will return instances matching the filter. In the case of `Win32_Directory`, a single object is returned, as the following output shows:

```
Name        Hidden  Archive  Writeable  LastModified
----        ------  -------  ---------  ------------
C:\Windows  False   False    True       20/02/2021 09:25:20
```

The `Filter` format is WQL and will be explored later in this chapter.

When returning large amounts of information, the `Property` parameter can be used to reduce the number of fields returned by a query:

```
Get-CimInstance Win32_UserAccount -Property Name, SID
```

You can also use the `Query` parameter, although it is rare to find a use for this that cannot be served by the individual parameters:

```
Get-CimInstance -Query "SELECT * FROM Win32_Process"
Get-CimInstance -Query "SELECT Name, SID FROM Win32_UserAccount"
```

Getting classes

The `Get-CimClass` command is used to return the details of a WMI class:

```
PS> Get-CimClass Win32_Process
NameSpace: ROOT/cimv2
CimClassName     CimClassMethods                CimClassProperties
------------     ---------------                ------------------
Win32_Process    {Create, Terminate, Get...}    {Caption, Description...}
```

The `Class` object describes the capabilities of that class. By default, `Get-CimClass` lists classes from the `root\cimv2` namespace.

The `Namespace` parameter will fill using tab completion, meaning if the following partial command is entered, pressing Tab repeatedly will cycle through the possible root namespaces:

```
Get-CimClass -Namespace <tab, tab, tab>
```

The child namespaces of a given namespace are listed in a `__Namespace` class instance. For example, the following command returns the namespaces under `root`:

```
Get-CimInstance __Namespace -Namespace root
```

Extending this technique, it is possible to recursively query `__Namespace` to find all the possible namespace values. Certain WMI namespaces are only available to administrative users (run as administrator); the following function may display errors for some namespaces:

```
function Get-CimNamespace {
    param (
        $Namespace = 'root'
    )

    Get-CimInstance __Namespace -Namespace $Namespace | ForEach-Object {
        $childNamespace = Join-Path -Path $Namespace -ChildPath $_.Name
        $childNamespace

        Get-CimNamespace -Namespace $childNamespace
    }
}
Get-CimNamespace
```

CIM class objects describe the methods available for a class. These methods are used to change objects.

Calling methods

You can use the `Invoke-CimMethod` command to call a method. The CIM class can be used to find details of the methods that a class supports:

```
PS> (Get-CimClass Win32_Process).CimClassMethods
Name        ReturnType   Parameters        Qualifiers
----        ----------   ----------        ----------
Create      UInt32       {CommandLine...}  {Constructor...}
Terminate   UInt32       {Reason}          {Destructor...}
GetOwner    UInt32       {Domain...}       {Implemented...}
GetOwnerSid UInt32       {Sid}             {Implemented...}
```

The method with the `Constructor` qualifier can be used to create a new instance of `Win32_Process`.

The `Parameters` property of a specific WMI method can be explored to find out how to use a method:

```
PS> (Get-CimClass Win32_Process).CimClassMethods['Create'].Parameters
Name                          CimType    Qualifiers
----                          -------    ----------
CommandLine                    String    {ID, In, MappingStrings}
CurrentDirectory               String    {ID, In, MappingStrings}
ProcessStartupInformation    Instance    {EmbeddedInstance, ID, In, MappingStrings}
ProcessId                      UInt32    {ID, MappingStrings, Out}
```

If an argument has the `In` qualifier, it can be passed in when you're creating an object. If an argument has the `Out` qualifier, it will be returned once the instance has been created. Arguments are passed in using a Hashtable.

When creating a process, the `CommandLine` argument is required; the rest can be ignored until later:

```
$params = @{
    ClassName  = 'Win32_Process'
    MethodName = 'Create'
    Arguments  = @{
        CommandLine = 'notepad.exe'
    }
}
$return = Invoke-CimMethod @params
```

The `return` object holds three properties in the case of the `Create` method of `Win32_Process`. This includes `ProcessId` and `ReturnValue`. PowerShell adds a `PSComputerName` property to these:

```
PS> $return
ProcessId     ReturnValue     PSComputerName
---------     -----------     --------------
    15172               0
```

`PSComputerName` is blank when a request is local. `ProcessId` is the `Out` property listed under the `Create` method parameters. `ReturnValue` indicates whether the operation succeeded, and `0` indicates that it was successful.

A nonzero return value indicates that something went wrong, but the values are not translated in PowerShell. The return values are documented in Microsoft Docs: https://docs.microsoft.com/windows/win32/cimwin32prov/create-method-in-class-win32-process.

The `Create` method used here creates a new instance. The other methods for `Win32_Process` act against an existing instance (an existing process).

Extending the preceding example, a process can be created and then terminated:

```
$params = @{
    ClassName  = 'Win32_Process'
    MethodName = 'Create'
    Arguments  = @{
        CommandLine = 'notepad.exe'
    }
}
$return = Invoke-CimMethod @params
pause
Get-CimInstance Win32_Process -Filter "ProcessID=$($return.ProcessId)" |
    Invoke-CimMethod -MethodName Terminate
```

The `pause` command will wait for return to be pressed before continuing; this allows us to show that Notepad was opened before it was terminated.

The `Terminate` method has an optional argument that is used as the exit code for the terminate process. This argument may be added using a Hashtable; in this case, a (made up) value of `5` is set as the exit code:

```
$invokeParams = @{
    ClassName  = 'Win32_Process'
    MethodName = 'Create'
    Arguments  = @{
        CommandLine = 'notepad.exe'
    }
}
$return = Invoke-CimMethod @invokeParams
$getParams = @{
    ClassName = 'Win32_Process'
    Filter    = 'ProcessId={0}' -f $return.ProcessId
}
Get-CimInstance @getParams |
    Invoke-CimMethod -MethodName Terminate -Arguments @{Reason = 5}
```

`Invoke-CimMethod` returns an object with a `ReturnValue`. A return value of `0` indicates that the command succeeded. A nonzero value indicates an error condition. The meaning of the value will depend on the WMI class.

The return values associated with the `Terminate` method of `Win32_Process` are documented in Microsoft Docs: https://docs.microsoft.com/windows/win32/cimwin32prov/terminate-method-in-class-win32-process.

Some methods require or can use instances of CIM classes as arguments. In some cases, these instances must be created.

Creating instances

The arguments for `Win32_Process` include a `ProcessStartupInformation` parameter. `ProcessStartupInformation` is

described by a WMI class, `Win32_ProcessStartup`.

There are no existing instances of `Win32_ProcessStartup`; running `Get-CimInstance Win32_ProcessStartup` will not find anything. The `Win32_ProcessStartup` class doesn't have a `Create` method (or any other constructor) that can be used to create an instance either.

You can use `New-CimInstance` to create an instance of the class:

```
$class = Get-CimClass Win32_ProcessStartup
$startupInfo = New-CimInstance -CimClass $class -ClientOnly
```

You can also use `New-Object`:

```
$class = Get-CimClass Win32_ProcessStartup
$startupInfo = New-Object CimInstance $class
```

Finally, the `new` method may be used:

```
$class = Get-CimClass Win32_ProcessStartup
$startupInfo = [CimInstance]::new($class)
```

You can set properties on the created instance; the effect of each property is documented in Microsoft Docs: https://docs.microsoft.com/windows/win32/cimwin32prov/win32-processstartup.

In the following example, properties are set to dictate the position and title of a `pwsh.exe` window:

```
$class = Get-CimClass Win32_ProcessStartup
$startupInfo = New-CimInstance -CimClass $class -ClientOnly
$startupInfo.X = 50
$startupInfo.Y = 50
$startupInfo.Title = 'This is the window title'
$params = @{
    ClassName  = 'Win32_Process'
    MethodName = 'Create'
    Arguments  = @{
        CommandLine                = 'pwsh.exe'
        ProcessStartupInformation = $startupInfo
    }
}
$returnObject = Invoke-CimMethod @params
```

If the process starts successfully, `$returnObject` will have a `ReturnValue` of `0`.

Working with CIM sessions

As we mentioned earlier in this chapter, a key feature of the CIM cmdlets is their ability to change how connections are formed and used.

The `Get-CimInstance` command has a `ComputerName` parameter, and when you use this, the command automatically creates a session to a remote system using `WSMAN`. The connection is destroyed as soon as the command completes.

The `Get-CimSession`, `New-CimSession`, `New-CimSessionOption`, and `Remove-CimSession` commands are optional commands that you can use to define the behavior of remote connections.

The `New-CimSession` command creates a connection to a remote server. An example is as follows:

```
PS> $cimSession = New-CimSession -ComputerName Remote1
PS> $cimSession
Id           : 1
Name         : CimSession1
InstanceId   : 1cc2a889-b649-418c-94a2-f24e033883b4
ComputerName : Remote1
Protocol     : WSMAN
```

Alongside the other parameters, `New-CimSession` has a `Credential` parameter that can be used in conjunction with `Get-Credential` to authenticate a connection.

If the remote system does not, for any reason, present access to `WSMAN`, it is possible to switch the protocol down to `DCOM` by using the `New-CimSessionOption` command:

```
PS> $option = New-CimSessionOption -Protocol DCOM
PS> $cimSession = New-CimSession -ComputerName Remote1 -SessionOption $option
PS> $cimSession
Id           : 2
Name         : CimSession2
InstanceId   : 62b2cb56-ec84-472c-a992-4bee59ee0618
ComputerName : Remote1
Protocol     : DCOM
```

The `New-CimSessionOption` command is not limited to protocol switching; it can affect many of the other properties of the connection, as shown in the help and the examples for the command.

Once a session has been created, it exists in memory until it is removed. The `Get-CimSession` command shows a list of connections that have been formed, and the `Remove-CimSession` command permanently removes connections.

Associated classes

The `Get-CimAssociatedClass` command replaces the use of the `ASSOCIATORS OF` query type when using the CIM cmdlets.

The following command gets the class instances associated with `Win32_NetworkAdapterConfiguration`. As the arguments for the `Get-CimInstance` command are long strings, splatting is used to pass the parameters into the command:

```
$params = @{
    ClassName = 'Win32_NetworkAdapterConfiguration'
    Filter    = 'IPEnabled=TRUE AND DHCPEnabled=TRUE'
}
Get-CimInstance @params | Get-CimAssociatedInstance
```

The following example uses `Get-CimAssociatedClass` to get the physical interface associated with the IP configuration:

```
$params = @{
    ClassName = 'Win32_NetworkAdapterConfiguration'
    Filter    = 'IPEnabled=TRUE AND DHCPEnabled=TRUE'
}
Get-CimInstance @params | ForEach-Object {
    $adapter = $_ | Get-CimAssociatedInstance -ResultClassName Win32_NetworkAdapter

    [PSCustomObject]@{
        NetConnectionID = $adapter.NetConnectionID
        Speed           = [Math]::Round($adapter.Speed / 1MB, 2)
        IPAddress       = $_.IPAddress
        IPSubnet        = $_.IPSubnet
        Index           = $_.Index
        Gateway         = $_.DefaultIPGateway
    }
}
```

The preceding command returns details of every IP- and DHCP-enabled network adapter, merging the results of two different CIM classes into a single object.

The WMI Query Language

WMI Query Language, or **WQL**, is used to query WMI in a similar style to SQL.

WQL implements a subset of **Structured Query Language** (**SQL**). The keywords are traditionally written in uppercase; however, WQL is not case-sensitive.

Both the CIM and the older WMI cmdlets support the `Filter` and `Query` parameters, which accept WQL queries.

Understanding SELECT, WHERE, and FROM

The `SELECT`, `WHERE`, and `FROM` keywords are used with the `Query` parameter.

The generalized syntax for the `Query` parameter is as follows:

```
SELECT <Properties> FROM <WMI Class>
SELECT <Properties> FROM <WMI Class> WHERE <Condition>
```

You can use the wildcard `*` to request all available properties or a list of known properties:

```
Get-CimInstance -Query "SELECT * FROM Win32_Process"
Get-CimInstance -Query "SELECT ProcessID, CommandLine FROM Win32_Process"
```

The `WHERE` keyword is used to filter results returned by `SELECT`; for example, see the following:

```
Get-CimInstance -Query "SELECT * FROM Win32_Process WHERE ProcessID=$PID"
```

WQL and arrays

WQL cannot filter array-based properties; for example, the capabilities property of `Win32_DiskDrive`.

Escape sequences and wildcards

The backslash character, `\`, is used to escape the meaning of characters in a WMI query. You can use it to escape a wildcard character, quotes, or itself. For example, the following WMI query uses a path; each instance of `\` in the path must be escaped:

```
Get-CimInstance Win32_Process -Filter "ExecutablePath='C:\\Windows\\Explorer.exe'"
```

The preceding command returns any instances of the `explorer.exe` process, as shown here:

```
ProcessId Name        HandleCount WorkingSetSize VirtualSize
--------- ----        ----------- -------------- -----------
8320      explorer.exe 3412       198606848      2204322111488
```

The properties shown will vary from one computer to another.

About Win32_Process and the Path property

The `Path` script property is added to the output from the `Win32_Process` class by PowerShell. While it appears in the output, the property cannot be used to define a filter, nor can `Path` be selected using the `Property` parameter of either `Get-CimInstance` or `Get-WmiObject`.

`Get-Member` shows that it is a `ScriptProperty`, as follows:

```
Get-CimInstance Win32_Process -Filter "ProcessId=$pid" |
    Get-Member -Name Path
Get-WmiObject Win32_Process -Filter "ProcessId=$pid" |
    Get-Member -Name Path
```

WQL defines two wildcard characters that can be used with string queries:

- The `%` (percentage) character matches any number of characters and is equivalent to using `*` in a filesystem path or with the `-like` operator.
- The `_` (underscore) character matches a single character and is equivalent to using `?` in a filesystem path or with the `-like` operator.

The following query filters the results of `Win32_Service`, including services with paths starting with a single drive letter and ending with `.exe`:

```
Get-CimInstance Win32_Service -Filter 'PathName LIKE "_:\\%.exe"'
```

Logic operators

Logic operators can be used with the `Filter` and `Query` parameters.

The examples in the following table are based on the following command:

```
Get-CimInstance Win32_Process -Filter "<Filter>"
```

Description	**Operator**	**Syntax**	**Example**
Logical and	`AND`	`<Condition1> AND <Condition2>`	`ProcessID=$pid AND Name='powershell.exe'`
Logical or	`OR`	`<Condition1> OR <Condition2>`	`ProcessID=$pid OR ProcessID=0`
Logical not	`NOT`	`NOT <Condition>`	`NOT ProcessID=$pid`

Table 11.1: WQL logical operators

Comparison operators

Comparison operators may be used with the `Filter` and `Query` parameters.

The examples in the following table are based on the following command:

```
Get-CimInstance Win32_Process -Filter "<Filter>"
```

Description	**Operator**	**Example**
Equal to	`=`	`Name='powershell.exe' AND ProcessId=0`
Not equal to	`<>`	`Name<>'powershell.exe'`
Greater than	`>`	`WorkingSetSize>$(100MB)`
Greater than or equal to	`>=`	`WorkingSetSize>=$(100MB)`
Less than	`<`	`WorkingSetSize<$(100MB)`
Less than or equal to	`<=`	`WorkingSetSize<=$(100MB)`
Is	`IS`	`CommandLine IS NULL` `CommandLine IS NOT NULL`

Like	`LIKE`	`CommandLine LIKE '%.exe'`

Table 11.2: WQL comparison operators

Quoting values

When building a WQL query, string values must be quoted; numeric and Boolean values do not need quotes.

As the filter is also a string, this often means nesting quotes within one another. The following techniques may be used to avoid needing to use PowerShell's escape character.

For filters or queries containing fixed string values, use either of the following styles. Use single quotes outside and double quotes inside:

```
Get-CimInstance Win32_Process -Filter 'Name="pwsh.exe"'
```

Alternatively, use double quotes outside and single quotes inside:

```
Get-CimInstance Win32_Process -Filter "Name='pwsh.exe'"
```

For filters or queries containing PowerShell variables or sub-expressions, use double quotes around the filter. Variables within single-quoted strings will not expand:

```
Get-CimInstance Win32_Process -Filter "ProcessId=$PID"
Get-CimInstance Win32_Process -Filter "ExecutablePath LIKE '$($pshome -replace '\\', '\\')%'"
```

Regex recap

The regular expression `'\\'` represents a single literal `'\'`, as the backslash is normally the escape character. Each `'\'` in the `pshome` path is replaced with `'\\'` to account for WQL using `'\'` as an escape character as well.

In the previous example, the filter extends over a single line. If a filter is long, or contains several conditions, consider using the format operator to compose the filter string:

```
$params = @{
    ClassName = 'Win32_Process'
    Filter    = "ExecutablePath LIKE '{0}%'" -f @(
        $PSHOME -replace '\\', '\\'
    )
}
Get-CimInstance @params
```

The format operator is used to add the escaped version of the path to the string.

The preceding command will show any instances of PowerShell that are running; for example:

```
ProcessId Name     HandleCount WorkingSetSize VirtualSize
--------- ----     ----------- -------------- -----------
14868     pwsh.exe 1114        243924992      2204239736832
10952     pwsh.exe 1513        180326400      2204260892672
```

WQL filters are especially useful when working with CIM classes that contain many instances.

Associated classes

WMI classes often have several different associated or related classes; for example, each instance of `Win32_Process` has an associated class, `CIM_DataFile`.

Associations between two classes are expressed by a third class. In the case of `Win32_Process` and `CIM_DataFile`, the relationship is expressed by the `CIM_ProcessExecutable` class.

The relationship is defined by using the antecedent and dependent properties, as shown in the following example:

```
PS> Get-CimInstance CIM_ProcessExecutable |
>>     Where-Object Dependent -match $PID |
>>     Select-Object -First 1
Antecedent          : CIM_DataFile (Name = "C:\WINDOWS\System32\Windo...)
Dependent           : Win32_Process (Handle = "11672")
BaseAddress         : 2340462460928
GlobalProcessCount  :
ModuleInstance      : 4000251904
ProcessCount        : 0
PSComputerName      :
```

This `CIM_ProcessExecutable` class does not need to be used directly; it is only used to express the relationship between two other classes.

WMI object paths

A WMI path is required to find classes associated with an instance. The WMI object path uniquely identifies a specific instance of a WMI class.

The object path is made up of several components:

```
<Namespace>:<ClassName>.<KeyName>=<Value>
```

The namespace can be omitted if the class is under the default namespace, `root\cimv2`.

You can discover the `KeyName` for a given WMI class in several ways. In the case of `Win32_Process`, the key name might be discovered by using any of the following methods.

It can be discovered by using the CIM cmdlets:

```
(Get-CimClass Win32_Process).CimClassProperties |
    Where-Object { $_.Flags -band 'Key' }
```

It can be discovered by using the Microsoft Docs website, which provides descriptions of each property (and method) exposed by the class: https://docs.microsoft.com/windows/win32/cimwin32prov/win32-process.

Having identified a key, only the value remains to be found. In the case of `Win32_Process`, `key` (handle) has the same value as the process ID. The object path for the `Win32_Process` instance associated with a running PowerShell console is, therefore, as follows:

```
root\cimv2:Win32_Process.Handle=$PID
```

The namespace does not need to be included if it uses the default, `root\cimv2`; the object path can be shortened to the following:

```
Win32_Process.Handle=$PID
```

`Get-CimInstance` and `Get-WmiObject` do not retrieve an instance from an object path, but the `Wmi` type accelerator can:

```
PS> [Wmi]"Win32_Process.Handle=$PID" | Select-Object Name, Handle
Name     Handle
----     ------
pwsh.exe 11672
```

The preceding object is somewhat equivalent to using a filter for `Handle` when querying `Win32_Process`.

Using ASSOCIATORS OF

Queries using `ASSOCIATORS OF` are used to find instances of classes that are related to a specific object.

The `ASSOCIATORS OF` query may be used for any given object path; for example, using the preceding object path results in the following command:

```
Get-CimInstance -Query "ASSOCIATORS OF {Win32_Process.Handle=$PID}"
```

This query will return objects from three different classes: `Win32_LogonSession`, `Win32_ComputerSystem`, and `CIM_DataFile`. The classes that are returned are shown in the following example:

```
PS> $params = @{
>>     Query = "ASSOCIATORS OF {Win32_Process.Handle=$PID}"
>> }
PS> Get-CimInstance @params | Select-Object CimClass -Unique
CimClass
--------
root/cimv2:Win32_ComputerSystem
root/cimv2:Win32_LogonSession
root/cimv2:CIM_DataFile
```

The query can be refined to filter a specific resulting class; an example is as follows:

```
Get-CimInstance -Query "ASSOCIATORS OF {Win32_Process.Handle=$PID} WHERE ResultClass = CIM_DATAFILE"
```

ResultClass values must not be quoted

The value in the `ResultClass` condition is deliberately not quoted.

The result of this operation is a long list of files that are used by the PowerShell process. A snippet of this is shown as follows:

```
PS> Get-CimInstance -Query "ASSOCIATORS OF {Win32_Process.Handle=$PID} WHERE ResultClass =
CIM_DATAFILE" |
>>     Select-Object Name
Name
----
C:\Program Files\PowerShell\7\pwsh.exe
C:\WINDOWS\SYSTEM32\ntdll.dll
C:\WINDOWS\System32\KERNEL32.DLL
C:\WINDOWS\System32\KERNELBASE.dll
C:\WINDOWS\System32\USER32.dll
C:\WINDOWS\System32\win32u.dll
```

While the older `Get-WmiObject` command has not been continued into PowerShell 6, the WMI type accelerators remain.

WMI Type Accelerators

The WMI cmdlets were removed in PowerShell 6 and are not going to be reinstated.

The following type accelerators remain and can still be used:

- `Wmi` : `System.Management.ManagementObject`
- `WmiClass` : `System.Management.ManagementClass`
- `WmiSearcher` : `System.Management.ManagementObjectSearcher`

When necessary, these accelerators may be used to simulate the functionality provided by the older WMI cmdlets.

Both the `Wmi` and `WmiClass` type accelerators can be written to use a remote computer by including the computer name. An example is as follows:

```
[Wmi]"\\RemoteComputer\root\cimv2:Win32_Process.Handle=$PID"
[WmiClass]"\\RemoteComputer\root\cimv2:Win32_Process"
```

You can use these classes in PowerShell 6 and higher.

Getting instances

You can use the type accelerator `WmiSearcher` to execute queries and retrieve results:

```
([WmiSearcher]"SELECT * FROM Win32_Process").Get()
```

The returned object is identical to the object that would have been returned by the `Get-WmiObject` command.

Working with dates

WMI instances retrieved using type accelerators do not convert date-time properties into the `DateTime` type. Querying the `Win32_Process` class for the creation date of a process returns the date-time property as a long string:

```
PS> $query = '
>> SELECT Name, CreationDate
>> FROM Win32_Process
>> WHERE ProcessId={0}
>> ' -f $PID
PS> ([WmiSearcher]$query).Get() | Select-Object Name, CreationDate
Name     CreationDate
----     ------------
pwsh.exe 20200510090416.263973+060
```

The .NET namespace, `System.Management`, includes a class called `ManagementDateTimeConverter`, dedicated to converting date and time formats found in WMI. This method is added to WMI objects in PowerShell as a `ConvertToDateTime` script method.

The string in the preceding example may be converted as follows:

```
$query = '
SELECT Name, CreationDate
FROM Win32_Process
WHERE ProcessId={0}
' -f $PID
([WmiSearcher]$query).Get() | Select-Object @(
    'Name'
    @{
        Name = 'CreationDate'
        Expression = {
            $_.ConvertToDateTime($_.CreationDate)
        }
    }
)
```

WMI classes may be used via the `WmiClass` accelerator.

Getting classes

An instance of a class may be created using the `WmiClass` accelerator, as the following code shows:

```
[WmiClass]'Win32_Process'
```

The class describes the methods and properties that can be used.

Calling methods

Calling a method on an existing instance of an object found using `WmiSearcher` is like using any other .NET method call.

The following example gets and restarts the "Print Spooler" service. The following operation requires administrative access:

```
$query = '
  SELECT *
  FROM Win32_SERVICE
  WHERE DisplayName="Print Spooler"
'
$service = ([WmiSearcher]$query).Get()
$service.StopService()     # Call the StopService method
$service.StartService()    # Call the StartService method
```

The WMI class can be used to find the details of a method; for example, the `Create` method of `Win32_Share`, as follows:

```
PS> ([WmiClass]'Win32_Share').Methods['Create']
Name          : Create
InParameters  : System.Management.ManagementBaseObject
OutParameters : System.Management.ManagementBaseObject
Origin        : Win32_Share
Qualifiers    : {Constructor, Implemented, MappingStrings, Static}
```

When the `Invoke-CimMethod` command accepts a Hashtable, methods invoked on a WMI object expect arguments to be passed in a specific order. The order is described in the documentation for the class, such as https://docs.microsoft.com/windows/win32/cimwin32prov/create-method-in-class-win32-share.

The documentation shows which arguments are mandatory (not optional) in the syntax element at the top.

Alternatively, the order arguments must be written like so:

```
PS> ([WmiClass]'Win32_Share').Create.OverloadDefinitions -split '(?<=,)'
System.Management.ManagementBaseObject Create(System.String Path,
 System.String Name,
 System.UInt32 Type,
 System.UInt32 MaximumAllowed,
 System.String Description,
 System.String Password,
 System.Management.ManagementObject#Win32_SecurityDescriptor Access)
```

To create a share, the argument list must contain an argument for `Access`, then `Description`, then `MaximumAllowed`, and so on. If the argument is optional, it can be ignored, or a null value may be provided:

```
([WmiClass]'Win32_Share').Create(
    'C:\Temp\Share1', # Path
    'Share2',         # Name
    0                 # Type (Disk Drive)
)
```

The `Description` argument follows the `MaximumAllowed` argument. If `Description` were to be set (but not `MaximumAllowed`), a null value can be added for that argument:

```
([WmiClass]'Win32_Share').Create(
    'C:\Temp\Share1', # Path
    'Share3',         # Name
    0,                # Type (Disk Drive),
    $null,            # MaximumAllowed
    'Description'     # Description
)
```

`ReturnValue` describes the result of the operation; a `ReturnValue` of `0` indicates success. As this operation requires administrator privileges (run as administrator), a `ReturnValue` of `2` is used to indicate that it was run without sufficient rights.

If the folder used in the previous example does not exist, `ReturnValue` will be set to `24`.

A less well-known alternative is available compared to passing arguments in an array. You can pass arguments by setting the values of an object. The object is retrieved using the `GetMethodParameters` method on a WMI class:

```
$class = [WmiClass]'Win32_Share'
$params = $class.GetMethodParameters('Create')
$params.Name = 'Share1'
$params.Path = 'C:\Temp\Share1'
$params.Type = 0
$class.InvokeMethod('Create', $params)
```

Creating an object to represent the parameters has a clear advantage in that each property has a clear name, rather than being reliant on discovering and using positional arguments.

Creating instances

An instance of a WMI class can be created using the `CreateInstance` method of the class. The following example creates an instance of `Win32_Trustee`:

```
([WmiClass]'Win32_Trustee').CreateInstance()
```

Associated classes

Objects returned by `WmiSearcher` have a `GetRelated` method that can be used to find associated instances.

The `GetRelated` method accepts arguments that can be used to filter the results. The first argument, `relatedClass`, is used to limit the instances that are returned to specific classes, as shown here:

```
([WmiSearcher]'SELECT * FROM Win32_LogonSession').Get() | ForEach-Object {
    [PSCustomObject]@{
        LogonName      = $_.GetRelated('Win32_Account').Caption
        SessionStarted = [System.Management.ManagementDateTimeConverter]::ToDateTime(
            $_.StartTime
        )
    }
}
```

Permissions

Working with permissions in WMI is more difficult than in .NET as the values in use are not given friendly names. However, the .NET classes can still be used, even if not quite as intended.

The following working examples demonstrate configuring the permissions.

Sharing permissions

`Get-Acl` and `Set-Acl` are fantastic tools for working with filesystem permissions, or permissions under other providers. However, these commands cannot be used to affect SMB share permissions.

The SmbShare module

The `SmbShare` module has commands that affect share permissions. This example uses the older WMI classes to modify permissions. It might be used if the `SmbShare` module cannot be.

The `Get-SmbShareAccess` command might be used to verify the outcome of this example.

The following operations require administrative privileges; use PowerShell as an administrator if you're attempting to use these examples.

Creating a shared directory

The following snippet creates a directory and shares that directory:

```
$path = 'C:\Temp\WmiPermissions'
New-Item $path -ItemType Directory
$params = @{
    ClassName = 'Win32_Share'
    MethodName = 'Create'
    Arguments = @{
        Name = 'WmiPerms'
        Path = $path
        Type = [UInt32]0
    }
}
Invoke-CimMethod @params
```

The `Create` method used here will fail if the argument for `Type` is not correctly defined as a `UInt32` value. PowerShell will set the type to `Int32` if the value of `0` is used without the cast.

The requirement for `UInt32`, in this case, may be viewed by exploring the parameters required for the method:

```
PS> (Get-CimClass Win32_Share).CimClassMethods['Create'].Parameters |
>>     Where-Object Name -eq Type
Name     CimType     Qualifiers                ReferenceClassName
----     -------     ----------                ------------------
Type     UInt32     {ID, In, MappingStrings}
```

Now that the share exists, you can retrieve the security descriptor assigned to the share.

Getting a security descriptor

When `Get-Acl` is used, the object that it gets is a security descriptor. The security descriptor includes a set of control information (ownership and so on), along with the discretionary and system access control lists.

The WMI class `Win32_LogicalShareSecuritySetting` is used to represent the security for each of the shares on a computer:

```
$params = @{
    ClassName = 'Win32_LogicalShareSecuritySetting'
    Filter    = "Name='WmiPerms'"
}
$security = Get-CimInstance @params
```

The returned object has a limited number of properties:

```
Caption        : Security settings of WmiPerms
Description    : Security settings of WmiPerms
SettingID      :
ControlFlags   : 32772
Name           : WmiPerms
PSComputerName :
```

This instance is important as it is required to use the `GetSecurityDescriptor` method:

```
$return = $security | Invoke-CimMethod -MethodName GetSecurityDescriptor
$aclObject = $return.Descriptor
```

The security descriptor held in the `aclObject` variable is different from the result returned by `Get-Acl`:

```
PS> $aclObject
ControlFlags   : 32772
DACL           : {Win32_ACE}
Group          :
Owner          :
SACL           :
TIME_CREATED   :
PSComputerName :
```

The `ControlFlags` value states that a DACL is present, and that the security descriptor information is stored in a contiguous block of memory. These values are described in Microsoft Docs: https://docs.microsoft.com/windows/win32/secauthz/security-descriptor-control.

The `DACL`, or **discretionary access control list**, is used to describe the permission levels for each security principal (a user, group, or computer account). Each entry in this list is an instance of `Win32_ACE`:

```
PS> $aclObject.DACL
AccessMask                : 1179817
AceFlags                  : 0
AceType                   : 0
GuidInheritedObjectType :
GuidObjectType            :
TIME_CREATED              :
Trustee                   : Win32_Trustee
PSComputerName            :
```

The `Win32_ACE` object has a `Trustee` property that holds the `Name`, `Domain`, and `SID` properties of the security principal (in this case, the `Everyone` principal):

```
PS> $aclObject.DACL.Trustee
Domain         :
Name           : Everyone
SID            : {1, 1, 0, 0...}
SidLength      : 12
SIDString      : S-1-1-0
TIME_CREATED   :
PSComputerName :
```

`AceFlags` describes how an ACE is to be inherited. As this is a share, the `AceFlags` property will always be `0`. Nothing can, or will, inherit this entry; .NET can be used to confirm this:

```
PS> [System.Security.AccessControl.AceFlags]0
None
```

`AceType` is either `AccessAllowed` (`0`) or `AccessDenied` (`1`). Again, .NET can be used to confirm this:

```
PS> [System.Security.AccessControl.AceType]0
AccessAllowed
```

Finally, the `AccessMask` property can be converted into a meaningful value with .NET as well. The access rights that can be granted on a share are a subset of those that might be assigned to a file or directory:

```
PS> [System.Security.AccessControl.FileSystemRights]1179817
ReadAndExecute, Synchronize
```

Putting this together, the entries in a shared `DACL` can be made much easier to understand:

```
using namespace System.Security.AccessControl

$aclObject.DACL | ForEach-Object {
    [PSCustomObject]@{
        Rights   = [FileSystemRights]$_.AccessMask
        Type     = [AceType]$_.AceType
        Flags    = [AceFlags]$_.AceFlags
        Identity = $_.Trustee.Name
    }
}
```

In the preceding example, the domain of the trustee is ignored. If the trustee is something other than `Everyone`, the domain should be included.

Adding an access control entry

To add an **Access Control Entry** (**ACE**) to an existing list, you must create a `Win32_ACE`. Creating an ACE requires a `Win32_Trustee`. The following `trustee` is created from the current user:

```
$trustee = New-CimInstance (Get-CimClass Win32_Trustee) -ClientOnly
$trustee.Domain = $env:USERDOMAIN
$trustee.Name = $env:USERNAME
```

`SID` does not need to be set on the `trustee` object, but if the security principal is invalid, attempting to apply the change to security will fail.

Then, you can create `Win32_ACE`. The following ACE grants full control of the share to `trustee`:

```
using namespace System.Security.AccessControl
$ace = New-CimInstance (Get-CimClass Win32_ACE) -ClientOnly
$ace.AccessMask = [UInt32][FileSystemRights]'FullControl'
$ace.AceType = [UInt32][AceType]'AccessAllowed'
$ace.AceFlags = [UInt32]0
$ace.Trustee = $trustee
```

The ACE can be added to the `DACL` using the `+=` operator:

```
$aclObject.DACL += $ace
```

Setting the security descriptor

Once the ACL has been changed, the modified security descriptor must be set. The instance returned by `Win32_LogicalShareSecuritySetting` contains a `SetSecurityDescriptor` method:

```
$params = @{
    MethodName = 'SetSecurityDescriptor'
    Arguments  = @{
        Descriptor = $aclObject
    }
}
$security | Invoke-CimMethod @params
```

A return value of `0` indicates that the change to the ACL has been successfully applied. You can view this change by looking at the properties of the share in File Explorer, as shown in the following example:

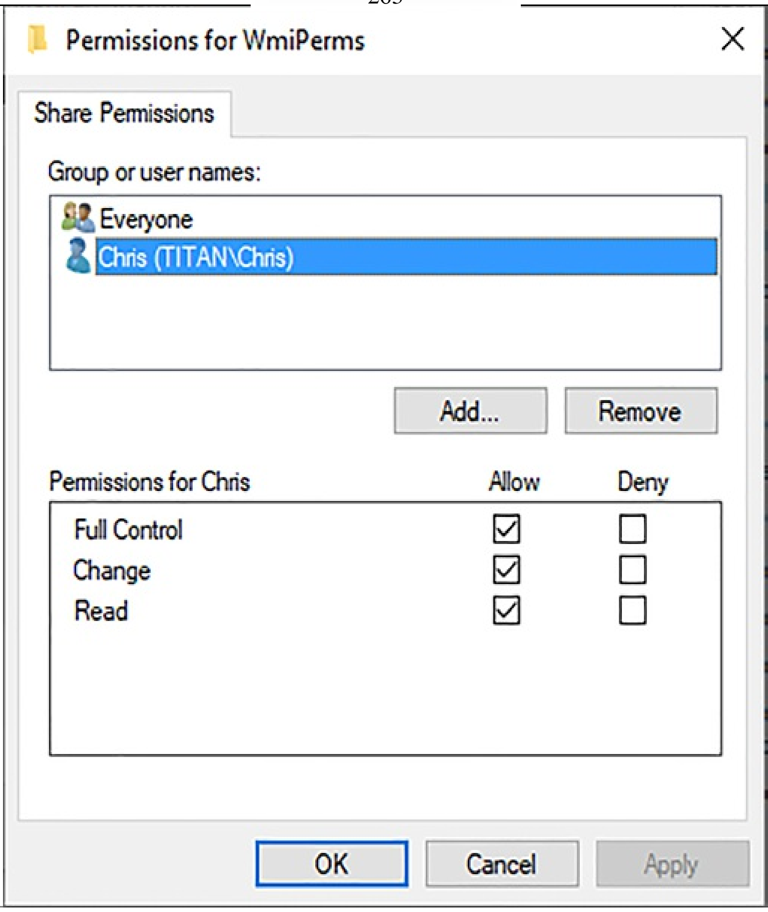


Figure 11.1: Modified share permissions

In early versions of PowerShell (and earlier versions of Windows), WMI was the only option for changing configurations like share permissions. The `SmbShare` module is an easier-to-use set of commands than the convoluted process used here. It is also able to show the assigned share permissions:

```
PS> Get-SmbShareAccess -Name WmiPerms
Name     ScopeName  AccountName  AccessControlType  AccessRight
----     ---------  -----------  -----------------  -----------
WmiPerms *          Everyone     Allow              Read
WmiPerms *          TITAN\Chris  Allow              Full
```

While share permissions have better commands available, the preceding process provides a good basis for modifying any security presented via WMI. One of these is permissions defined on namespaces in WMI.

WMI permissions

Getting and setting WMI security in PowerShell uses the same approach as share security. WMI permissions might be set using `wmimgmt.msc` if the GUI is used. The content of the `DACL` differs slightly.

The `__SystemSecurity` class is used to access the security descriptor. Each WMI namespace has its own instance of the `__SystemSecurity` class; an example is as follows:

```
Get-CimClass __SystemSecurity -Namespace root
Get-CimClass __SystemSecurity -Namespace root\cimv2
```

Getting a security descriptor

The security descriptor for a given namespace can be retrieved from the `__SystemSecurity` class. By default, administrator privileges are required to get the security descriptor:

```
$security = Get-CimInstance __SystemSecurity -Namespace root\cimv2
$return = $security | Invoke-CimMethod -MethodName GetSecurityDescriptor
$aclObject = $return.Descriptor
```

The access mask

The access mask defines which rights are assigned by the ACE. The values of an access mask in the `DACL` are documented as access right constants in Microsoft Docs: https://docs.microsoft.com/windows/win32/wmisdk/namespace-access-rights-constants.

The standard access rights, `ReadSecurity` and `WriteSecurity`, are also relevant. The access mask is a composite of the values listed here:

- `EnableAccount`: `1`
- `ExecuteMethods`: `2`
- `FullWrite`: `4`
- `PartialWrite`: `8`
- `WriteProvider`: `16`
- `RemoteEnable`: `32`
- `ReadSecurity`: `131072`
- `WriteSecurity`: `262144`

You can use these values to add a new ACE and set the security descriptor in the same way, just like when setting permissions on a share in the previous section.

The WMI methods used in this subsection also allow a security descriptor to be exported to and imported from a **Security Descriptor Definition Language** (**SDDL**) string.

WMI and SDDL

SDDL is used to describe the content of a security descriptor as a string. The string format is described in Microsoft Docs: https://docs.microsoft.com/windows/win32/secauthz/security-descriptor-string-format.

A security descriptor returned by `Get-Acl` has a method that can convert the entire security descriptor into a string, as follows:

```
PS> (Get-Acl C:\).GetSecurityDescriptorSddlForm('All')
O:S-1-5-80-956008885-3418522649-1831038044-1853292631-2271478464G:S-1-5-80-956008885-3418522649-
1831038044-1853292631-2271478464D:PAI(A;;LC;;;AU)(A;OICIIO;SDGXGWGR;;;AU)(A;;FA;
;;SY)(A;OICIIO;GA;;;SY)(A;OICIIO;GA;;;BA)(A;;FA;;;BA)(A;OICI;0x1200a9;;;BU)
```

A security descriptor defined using `SDDL` can also be imported. If the `sddlString` variable is assumed to hold a valid security descriptor, then you can use the following command:

```
$acl = Get-Acl C:\
$acl.SetSecurityDescriptorSddlForm($sddlString)
```

The imported security descriptor will not apply to the directory until `Set-Acl` is used.

WMI security descriptors can be converted to and from different formats, including `SDDL`. WMI has a specialized class for this: `Win32_SecurityDescriptorHelper`. The methods for this class are shown here:

```
PS> (Get-CimClass Win32_SecurityDescriptorHelper).CimClassMethods
Name                ReturnType  Parameters              Qualifiers
----                ----------  ----------              ----------
Win32SDToSDDL           UInt32  {Descriptor, SDDL}      {implemented, static}
Win32SDToBinarySD       UInt32  {Descriptor, BinarySD}  {implemented, static}
SDDLToWin32SD           UInt32  {SDDL, Descriptor}      {implemented, static}
SDDLToBinarySD          UInt32  {SDDL, BinarySD}        {implemented, static}
BinarySDToWin32SD       UInt32  {BinarySD, Descriptor}  {implemented, static}
BinarySDToSDDL          UInt32  {BinarySD, SDDL}        {implemented, static}
```

You might want to convert a WMI security descriptor into `SDDL` to create a backup before it makes a change, as follows:

```
$security = Get-CimInstance __SystemSecurity -Namespace root\cimv2
$return = $security | Invoke-CimMethod -MethodName GetSecurityDescriptor
$aclObject = $return.Descriptor
$params = @{
    ClassName = 'Win32_SecurityDescriptorHelper'
    MethodName = 'Win32SDToSDDL'
    Arguments  = @{
        Descriptor = $aclObject
    }
}
$return = Invoke-CimMethod @params
```

If the operation succeeds (that is, if `ReturnValue` is `0`), the security descriptor in `SDDL` form will be available:

```
PS> $return.SDDL
O:BAG:BAD:AR(A;CI;CCDCWP;;;S-1-5-21-2114566378-1333126016-908539190-1001)(A;CI;CCDCLCSWRPWPRCWD;;;BA)
(A;CI;CCDCRP;;;NS)(A;CI;CCDCRP;;;LS)(A;CI;CCDCRP;;;AU)
```

You can import a security descriptor expressed as an `SDDL` string:

```
$params = @{
    ClassName = 'Win32_SecurityDescriptorHelper'
    MethodName = 'SDDLToWin32SD'
    Arguments  = @{
        SDDL = 'O:BAG:BAD:AR(A;CI;CCDCWP;;;S-1-5-21-2114566378-1333126016-908539190-1001)
(A;CI;CCDCLCSWRPWPRCWD;;;BA)(A;CI;CCDCRP;;;NS)(A;CI;CCDCRP;;;LS)(A;CI;CCDCRP;;;AU)'
    }
}
$return = Invoke-CimMethod @params
$aclObject = $return.Descriptor
```

If `ReturnValue` is `0`, the `aclObject` variable will contain the imported security descriptor:

```
PS> $aclObject
ControlFlags   : 33028
DACL           : {Win32_ACE, Win32_ACE, Win32_ACE, Win32_ACE...}
Group          : Win32_Trustee
Owner          : Win32_Trustee
SACL           :
TIME_CREATED   :
PSComputerName :
```

The ACL object, like much of WMI, is not particularly descriptive. The raw information is there, and the descriptor can be set. Reading the descriptor requires work, as you must translate all the values into human-readable equivalents.

Summary

This chapter explored working with WMI classes, the different available commands, and the WMI Query Language. You used CIM cmdlets as a means of working with WMI.

Since `Get-WmiObject` has been removed from PowerShell, the WMI type accelerators were explored as an alternative means of working with WMI. This may be useful for the few rare classes that do not work using the CIM cmdlets. You used getting and setting permissions with WMI while using shared security and WMI security as examples.

Chapter 12, *Working with HTML, XML, and JSON*, will explore working with and generating and consuming data from a variety of different text-based formats.

12

Working with HTML, XML, and JSON

PowerShell has several commands for working with HTML, XML, and **JavaScript Object Notation** (**JSON**). These commands, combined with some of the available .NET classes, provide a rich set of tools for creating or modifying these formats.

This chapter covers the following topics:

- HTML
- XML commands
- `System.Xml`
- `System.Xml.Linq`
- JSON

HTML

PowerShell includes the `ConvertTo-Html` command, which can generate HTML content.

You can create more complex HTML documents using the `PSWriteHtml` module available in the PowerShell Gallery, which uses a **Domain-Specific Language** (**DSL**) to describe HTML documents in PowerShell.

> **What is a Domain-Specific Language?**
>
> A Domain-Specific Language or DSL is a specialized language used to describe something, in this case, an HTML document. Keywords are used to describe individual elements of the document. In this case, those keywords are hierarchically arranged.
>
> Other DSLs frequently used in PowerShell include **Desired State Configuration** (**DSC**) and Pester.

ConvertTo-Html

`ConvertTo-Html` generates an HTML document with a table based on an input object. The following example generates a table based on the output from `Get-Process`:

```
Get-Process | ConvertTo-Html -Property Name, Id, WorkingSet
```

The preceding code generates a table format by default. `Table` is the default value for the `-As` parameter. The `-As` parameter also accepts `List` as an argument to generate output like `Format-List`.

Multiple tables

You can use `ConvertTo-Html` to build more complex documents by using the `Fragment` parameter. The `Fragment` parameter generates an HTML table or list only (instead of a full document). Tables can be combined to create a larger document:

```
# Create the body
$body = @(
    '<h1>Services</h1>'
    Get-Service |
        Where-Object Status -eq 'Running' |
        ConvertTo-Html -Property Name, DisplayName -Fragment
    '<h1>Processes</h1>'
    Get-Process |
        Where-Object WorkingSet -gt 50MB |
        ConvertTo-Html -Property Name, Id, WorkingSet-Fragment
) | Out-String
# Create a document with the merged body
ConvertTo-Html -Body $body -Title Report |
    Set-Content report.html
```

Adding style

You can enhance HTML content by adding a **Cascading Style Sheet** (**CSS**) fragment. When CSS is embedded in an HTML document, it is added between style tags in the head element.

The following style uses CSS to change the font, color the table headers, define the table borders, and justify the table content:

```
$css = @'
<style>
    body { font-family: Arial; }
    table {
        width: 100%;
        border-collapse: collapse;
    }
    table, th, td {
        border: 1px solid Black;
        padding: 5px;
    }
    th {
        text-align: left;
        background-color: LightBlue;
    }
    tr:nth-child(even) {
        background-color: GainsBoro;
    }
</style>
'@
```

The `Head` parameter of `ConvertTo-Html` is used to add the element to the document:

```
Get-Process |
    ConvertTo-Html -Property Name, Id, WorkingSet -Head $css |
    Set-Content report.html
```

The CSS language is complex and very capable. The elements that are used in the preceding code, and many more, are documented with examples on the W3Schools website: https://www.w3schools.com/css/.

Different browsers support different parts of the CSS language, and email clients tend to support a smaller set still. Testing in the expected client is an important part of developing content.

> **ConvertTo-Html and Send-MailMessage**
>
> `ConvertTo-Html` outputs an array of strings, while `Send-MailMessage` will only accept a body as a string. Attempting to use the output from `ConvertTo-Html` with `Send-MailMessage` directly will raise an error.

The `Out-String` command may be added to ensure the output from `ConvertTo-Html` is a string:

```
$messageBody = Get-Process |
    ConvertTo-Html Name, Id, WorkingSet -Head $css |
    Out-String
```

HTML and special characters

HTML defines a number of special characters; for example, a literal ampersand (`&`) in HTML must be written as `&` .

`ConvertTo-Html` will handle the conversion of special characters in input objects, but it will not work with special characters in raw HTML that are added using the `Body` , `Head` , `PreContent` , or `PostContent` parameters.

For example, the `PreContent` parameter might be used in place of inline HTML in the *Multiple tables* example:

```
# Create the body
$body = @(
    Get-Service |
        Where-Object Status -eq 'Running' |
        ConvertTo-Html -PreContent '<h1>Services</h1>' -Property @(
            'Name'
            'DisplayName'
        ) -Fragment

    Get-Process |
        Where-Object WorkingSet -gt 50MB |
        ConvertTo-Html -PreContent '<h1>Processes</h1>' -Property @(
            'Name'
            'Id'
            'WorkingSet'
        ) -Fragment
) | Out-String
# Create a document with the merged body
ConvertTo-Html -Body $body -Title Report |
    Set-Content report.html
```

The `System.Web.HttpUtility` class includes methods that are able to convert strings containing such characters.

Before `System.Web.HttpUtility` can be used, the assembly must be added:

```
Add-Type -AssemblyName System.Web
```

The `HtmlEncode` static method will take a string and replace any reserved characters with HTML code. For example, the following snippet will replace `>` with `>` :

```
PS>'<h1>{0}</h1>' -f [System.Web.HttpUtility]::HtmlEncode('Files > 100MB')
<h1>Files &gt; 100MB</h1>
```

The `HtmlDecode` static method can be used to reverse the process:

```
PS> [System.Web.HttpUtility]::HtmlDecode("<h1>Files &gt; 100MB</h1>")
<h1>Files > 100MB</h1>
```

The `ConvertTo-Html` command can be used to very quickly build HTML documents; these can be sent as email content, or used as web pages.

HTML is one of several text-based formats supported by PowerShell and XML is another (working with XML content is a common requirement within PowerShell).

XML commands

Extensible Markup Language (**XML**) is a plain text format that is used to store structured data. XML is written to be both human and machine readable.

PowerShell includes `Select-Xml` and `ConvertTo-Xml` commands to work with XML content.

Before exploring the commands, let's look at the basic structure of XML documents.

About XML

XML documents often begin with a declaration, as shown here:

```
<?xml version="1.0" encoding="utf-8"?>
```

This declaration has three possible attributes. The version attribute is mandatory when a declaration is included:

- `version` : The XML version, `1.0` or `1.1`
- `encoding` : The file encoding, most frequently `utf-8` or `utf-16`
- `standalone` : Whether the XML file uses an internal or external **Document Type Definition** (**DTD**); permissible values are `yes` or `no`

The use of the standalone directive with DTD is beyond the scope of this chapter.

Elements and attributes

XML is similar in appearance to HTML. Elements begin and end with a tag name. The tag name describes the name of an element, for example:

```
<?xml version="1.0"?>
<rootElement>value</rootElement>
```

An XML document can only have one root element, but an element may have many descendants:

```
<?xml version="1.0"?>
<rootElement>
    <firstChild>1</firstChild>
    <secondChild>2</secondChild>
</rootElement>
```

An element may also have attributes. The `rootElement` element in the following example has an attribute named `attr` :

```
<?xml version="1.0"?>
<rootElement attr="value">
   <child>1</child>
</rootElement>
```

Any XML document may include references to one or more schemas. When more than one schema is in use, the schemas are applied to a document (or parts of a document) using a namespace attribute.

Namespaces

XML documents can use one or more namespaces. A namespace is normally used to associate a node or set of nodes with a specific XML schema.

XML namespaces are declared in an attribute with a name prefixed by `xmlns:` , for example:

```xml
<?xml version="1.0"?>
<rootElement xmlns:item="http://namespaces/item">
    <item:child>1</item:child>
</rootElement>
```

The XML namespace uses a URL as a unique identifier. The identifier is used to describe an element as belonging to a schema.

Schemas

An XML schema can be used to describe and constrain the elements, attributes, and values within an XML document.

> **About DTD**
>
> A document type definition, or DTD, may be used to constrain the content of an XML file. As a DTD has little bearing on the use of XML in PowerShell, it is considered beyond the scope of this book.

XML schema definitions are saved with an XSD extension. Schema files can be used to validate the content of an XML file.

The following is a simple schema that defines the elements and values permissible in the item namespace of the previous XML document:

```xml
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://namespaces/item"
           xmlns="https://www.w3schools.com"
           elementFormDefault="qualified">
    <xs:element name="rootElement">
        <xs:element name="child" type="xs:string" />
    </xs:element>
</xs:schema>
```

Schemas are explored again later in this chapter when validating a document or inferring a schema from an existing document.

Select-Xml

The `Select-Xml` command may be used to search XML documents using the `XPath` query language. PowerShell (and .NET) uses `XPath 1.0`.

> **More about XPath**
>
> The structure and format of `XPath` queries are beyond the scope of this chapter. However, web resources are available, including the reference on Microsoft Docs: https://docs.microsoft.com/en-us/previous-versions/dotnet/netframework-4.0/ms256115(v=vs.100)

Function names, element names, and values used in `XPath` queries, and XML in general, are case-sensitive.

Given the following XML snippet, `Select-Xml` might use an `XPath` expression to select the engines of green cars:

```
$string = @"
<?xml version="1.0"?>
<cars>
    <car type="Saloon">
        <colour>Green</colour>
        <doors>4</doors>
        <transmission>Automatic</transmission>
        <engine>
            <size>2.0</size>
            <cylinders>4</cylinders>
        </engine>
    </car>
</cars>
"@
```

The `-XPath` parameter and the result are shown here:

```
PS> Select-Xml -XPath '//car[colour="Green"]/engine' -Content $string |
    Select-Object -ExpandProperty Node

size    cylinders
----    ---------
2.0     4
```

When the XML document includes namespaces, `Select-Xml` must include the `-Namespace` parameter.

Select-Xml and namespaces

Each of the namespaces used in an XPath expression used to search a document must be included in the `-Namespace` parameter as a Hashtable when performing a search:

```
[Xml]$xml = @"
<?xml version="1.0"?>
<cars xmlns:c="http://example/cars">
    <car type="Saloon">
        <c:colour>Green</c:colour>
        <c:doors>4</c:doors>
        <c:transmission>Automatic</c:transmission>
        <c:engine>
            <size>2.0</size>
            <cylinders>4</cylinders>
        </c:engine>
    </car>
</cars>
"@
Select-Xml '//car' -Xml $xml -Namespace @{
    c = 'http://example/cars'
}
```

In the preceding example, the document uses the prefix `c` to refer to the `http://example/cars` namespace. This prefix is repeated in the XPath search, and in the `-Namespace` parameter for `Select-Xml`.

The prefix itself is, in this context, unimportant. The ID, or the **Uniform Resource Identifier** (**URI**), is the critical part. The prefix value can therefore be changed in the search, provided that the new prefix value uses the same URI.

```
[Xml]$xml = @"
<?xml version="1.0"?>
<cars xmlns:c="http://example/cars">
    <car type="Saloon">
        <c:colour>Green</c:colour>
        <c:doors>4</c:doors>
        <c:transmission>Automatic</c:transmission>
        <c:engine>
            <size>2.0</size>
            <cylinders>4</cylinders>
        </c:engine>
    </car>
</cars>
"@
Select-Xml '//car' -Xml $xml -Namespace @{
    x = 'http://example/cars'
}
```

The result of the search will show the prefix used in the search, not the prefix from the XML document:

```
Node   Path        Pattern
----   ----        -------
engine InputStream //car/x:engine
```

`Select-Xml` is useful for finding content within an XML document. The `ConvertTo-Xml` command may be useful when creating an XML representation of an object.

ConvertTo-Xml

The `ConvertTo-XML` command creates an XML representation of an object as an `XmlDocument`. For example, the current PowerShell process object might be converted into XML:

```
Get-Process -Id $pid | ConvertTo-Xml
```

XML is text

The command used in the previous example creates an XML representation of the object. All numeric values are stored as strings. The following example shows that the `WorkingSet` property, an integer, is held as a string:

```
$xml = Get-Process -Id $pid | ConvertTo-Xml
$property = $xml.Objects.Object.Property |
    Where-Object Name -eq WorkingSet
$property.'#text'.GetType()
```

The two commands explored in this section are handy, but the commands don't account for making changes to documents. When changing documents, the `System.Xml.XmlDocument` type is frequently used.

System.Xml

PowerShell primarily uses the `System.Xml.XmlDocument` type to work with XML content. This type is part of the `System.Xml` namespace. The documentation for the types available in this namespace is available on Microsoft Docs: https://docs.microsoft.com/dotnet/api/system.xml.

The `System.Xml.XmlDocument` type is normally used via a type accelerator.

The XML type accelerator

The XML type accelerator (`[Xml]`) can be used to create instances of `XmlDocument` , as shown in the following code:

```
[Xml]$xml = @"
<?xml version="1.0"?>
<cars>
    <car type="Saloon">
        <colour>Green</colour>
        <doors>4</doors>
        <transmission>Automatic</transmission>
        <engine>
            <size>2.0</size>
            <cylinders>4</cylinders>
        </engine>
    </car>
</cars>
"@
```

Elements and attributes of an `XmlDocument` object may be accessed as if they were properties. This is a feature of the PowerShell language rather than the .NET object:

```
PS> $xml.cars.car

type         : Saloon
colour       : Green
doors        : 4
transmission : Automatic
engine       : engine
```

If the document contains more than one car element, each of the instances will be returned. Content may be filtered using `Where-Object` if required, for example:

```
[Xml]$xml = @"
<?xml version="1.0"?>
<cars>
    <car type="Saloon">
        <colour>Green</colour>
    </car>
    <car type="Saloon">
        <colour>Blue</colour>
    </car>
</cars>
"@
$xml.cars.car | Where-Object type -eq 'Saloon'
```

Treating the XML document as a PowerShell object makes opening the document very convenient and requires no specialist knowledge of the `XmlDocument` type.

Searching large documents using `Where-Object` can be slow. For larger documents, XPath is more appropriate.

XPath and XmlDocument

The `Select-Xml` command introduced earlier in this chapter is one way of searching a document. If the document is held in a variable with the expectation that changes are to be made, then you can use the `SelectNodes` and `SelectSingleNodes` methods.

The following `SelectNodes` method is used with an XPath expression to find the `engine` node for all `car` elements where the value of the `colour` element is `Green` .

```
[Xml]$xml = @"
<?xml version="1.0"?>
<cars>
    <car type="Saloon">
        <colour>Green</colour>
        <doors>4</doors>
        <transmission>Automatic</transmission>
        <engine>
            <size>2.0</size>
            <cylinders>4</cylinders>
        </engine>
    </car>
</cars>
"@
$xml.SelectNodes('//car[colour="Green"]/engine')
```

The `SelectNodes` method is appropriate when more than one element might be returned (or an unknown number is expected).

SelectNodes and XPathNodeList

If the `SelectNodes` method is called, and there are no results, an empty `XPathNodeList` object is returned. The following condition is flawed:

```
$nodes = $xml.SelectNodes('//car[colour="Blue"]')
if ($nodes) {
    Write-Host "A blue car record exists"
}
```

In this case, using the `Count` property is a better approach. The condition below will only succeed if the document contains at least one matching node:

```
if ($nodes.Count -ge 1) {
    Write-Host "A blue car record exists"
}
```

If a single element is expected from a search, you can use the `SelectSingleNodes` method instead:

```
[Xml]$xml = @"
<?xml version="1.0"?>
<cars>
    <car type="Saloon">
        <numberPlate>abcd</numberPlate>
        <colour>Green</colour>
        <doors>4</doors>
        <transmission>Automatic</transmission>
        <engine>
            <size>2.0</size>
            <cylinders>4</cylinders>
        </engine>
    </car>
</cars>
"@
$xml.SelectSingleNode('//car[numberPlate="abcd"]')
```

This is useful when retrieving an element with a specific identity, where the match will be unique.

XML documents often use namespaces to indicate that certain elements belong to a specific schema. These namespaces must be provided when searching a document.

Working with namespaces

If an XML document includes a namespace, then queries for elements within the document are more difficult. Not only must the namespace tag be included, but `XmlNamespaceManager` must be defined. `XmlNamespaceManager` is used to describe the different namespaces in use in the document.

If you're using the `SelectNodes` method, `XmlNamespaceManager` must be built first and passed as an argument. `XmlNamespaceManager` itself requires the `NameTable` property from the XML document as an argument.

```
$namespaceManager = [System.Xml.XmlNamespaceManager]::new(
    $xml.NameTable
)
$namespaceManager.AddNamespace('c', 'http://example/cars')
$xml.SelectNodes(
    '//car[c:colour="Green"]/c:engine',
    $namespaceManager
)
```

XML documents, such as group policy reports, are difficult to work with as they often contain many different namespaces. Each of the possible namespaces must be added to a namespace manager for a search using those namespaces to be successful.

Creating XML documents

PowerShell can be used to create XML documents from scratch. One possible way to do this is by using the `XmlWriter` class. An instance of the `XmlWriter` class is created using the `Create` static method:

```
$writer = [System.Xml.XmlWriter]::Create("$pwd\newfile.xml")
$writer.WriteStartDocument()
$writer.WriteStartElement('cars')
$writer.WriteStartElement('car')
$writer.WriteAttributeString('type', 'Saloon')
$writer.WriteElementString('colour', 'Green')
$writer.WriteEndElement()
$writer.WriteEndElement()
$writer.Flush()
$writer.Close()
```

The use of the `Create` method of this type is described on Microsoft Docs: https://docs.microsoft.com/dotnet/api/system.xml.xmlwriter.create.

Elements opened by `WriteStartElement` must be closed to maintain a consistent document.

The `XmlWriter` class is a buffered writer; content is not immediately written, it is held in a buffer and periodically flushed or pushed to the underlying file (or stream). The `Flush` method is called at the end to ensure all content of the buffer has been written to the file.

The format of generated XML can be changed by supplying an `XmlWriterSettings` object when calling the `Create` method. For example, it might be desirable to write line breaks and indent elements, as shown in the following example:

```
$writerSettings = [System.Xml.XmlWriterSettings]@{
    Indent = $true
}
$writer = [System.Xml.XmlWriter]::Create(
    "$pwd\newfile.xml",
    $writerSettings
)
$writer.WriteStartDocument()
$writer.WriteStartElement('cars')
$writer.WriteStartElement('car')
$writer.WriteAttributeString('type', 'Saloon')
$writer.WriteElementString('colour', 'Green')
$writer.WriteEndElement()
$writer.WriteEndElement()
$writer.Flush()
$writer.Close()
```

The preceding example creates a short XML document in a file named `newfile.xml`. The following shows the content of the file:

```
<?xml version="1.0" encoding="utf-8"?>
<cars>
  <car type="Saloon">
    <colour>Green</colour>
  </car>
</cars>
```

Modifying an existing document is a common requirement. A document can be changed in several ways.

Modifying element and attribute values

Existing elements in an XML document can be modified by assigning a new value. For example, the misspelling of `Appliances` could be corrected:

```
[Xml]$xml = @"
<?xml version="1.0"?>
<items>
    <item name='Fridge'>
        <category>Appliances</category>
    </item>
    <item name='Cooker'>
        <category>Appliances</category>
    </item>
</items>
"@
($xml.items.item | Where-Object name -eq 'Fridge').category = 'Appliances'
```

Attributes may be changed in the same way; the interface does not distinguish between the two value types.

A direct assignment of a new value cannot be used if the XML document contains more than one element or attribute with the same name (at the same level). For example, the following XML snippet has two values with the same name:

```
[Xml]$xml = @"
<?xml version="1.0"?>
<list>
    <name>one</name>
    <name>two</name>
</list>
"@
```

The first value may be changed if it is uniquely identified and selected:

```
$xml.list.SelectSingleNode('./name[.="one"]').'#text' = 'three'
```

The following example shows a similar change being made to the value of an attribute:

```
[Xml]$xml = @"
<?xml version="1.0"?>
    <list name='letters'>
    <name>1</name>
</list>
"@
$xml.SelectSingleNode('/list[@name="letters"]').
    SetAttribute('name', 'numbers')
```

The `@` symbol preceding `name` in the `XPath` expression denotes that the value type is an attribute. If the attribute referred to by the `SetAttribute` method does not exist, it will be created.

Adding elements

Elements must be created before they can be added to an existing document. Elements are created in the context of a document:

```
[Xml]$xml = @"
<?xml version="1.0"?>
<list type='numbers'>
    <name>1</name>
</list>
"@
$newElement = $xml.CreateElement('name')
$newElement.InnerText = 2
$xml.list.AppendChild($newElement)
```

Complex elements may be built up by repeatedly using the `Create` method of the `XmlDocument` (held in the `$xml` variable).

If the new node is substantial, it may be easier to treat the new node set as a separate document and merge one into the other.

Removing elements and attributes

Elements may be removed from a document by selecting the node, then calling the `RemoveChild` method on the parent:

```
[Xml]$xml = @"
<?xml version="1.0"?>
<list type='numbers'>
    <name>1</name>
    <name>2</name>
    <name>3</name>
</list>
"@
$node = $xml.SelectSingleNode('/list/*[.="3"]')
$null = $node.ParentNode.RemoveChild($node)
```

The `RemoveAll` method is also available; however, this removes all children (and attributes) of the selected node.

Attributes are also easy to remove from a document:

```
$xml.list.RemoveAttribute('type')
```

At times it is necessary to copy content between two different XML documents.

Copying nodes between documents

Nodes (elements, attributes, and so on) can be copied and moved between different XML documents. To bring a node from an external document into another, you must first import it.

The following example creates two simple XML documents. The first (the `xml` variable) is the intended destination. The `newNodes` variable contains a set of elements that should be copied:

```
[Xml]$xml = @"
<?xml version="1.0"?>
<list type='numbers'>
    <name>1</name>
</list>
"@
[Xml]$newNodes = @"
<root>
    <name>2</name>
    <name>3</name>
    <name>4</name>
</root>
"@
```

Copying the `name` nodes requires each node to be selected in turn, imported into the original document, and added to the desired node:

```
foreach ($node in $newNodes.SelectNodes('/root/name')) {
    $newNode = $xml.ImportNode($node, $true)
    $null = $xml.list.AppendChild($newNode)
}
```

The `ImportNode` method requires two parameters: the node from the foreign document (`newNodes`) and whether the import is deep (one level or fully recursive).

The resulting XML can be viewed by inspecting the `OuterXml` property of the `xml` variable:

```
PS> $xml.OuterXml
<?xml version="1.0"?><list type="numbers"><name>1</name><name>2</name><name>3</name><name>4</name>
</list>
```

Schema validation

XML documents that reference a schema can be validated against that schema. The schema defines which elements and attributes may be present, the values and value types, and even the order elements can appear.

Windows PowerShell comes with several XML files with associated schema in the help files. For example, here's the help file for `ISE`:

```
PS> Get-Item $env:windir\System32\WindowsPowerShell\v1.0\modules\ISE\en-US\ISE-help.xml

    Directory: C:\Windows\System32\WindowsPowerShell\v1.0\modules\ISE\en-US

Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a----         29/11/16     07:57          33969 ISE-help.xml
```

This file has been chosen as it will fail schema validation.

The schema documents used by the help content are saved in `C:\Windows\System32\WindowsPowerShell\v1.0\Schemas\PSMaml`.

The following snippet may be used to load the schema files and then test the content of the document:

```
$path = 'C:\Windows\System32\WindowsPowerShell\v1.0\modules\ISE\en-US\ISE-help.xml'
$document = [Xml]::new()
$document.Load($path)
$document.Schemas.XmlResolver = [System.Xml.XmlUrlResolver]::new()
$document.Schemas.Add(
    'http://schemas.microsoft.com/maml/2004/10',
    'C:\Windows\System32\WindowsPowerShell\v1.0\Schemas\PSMaml\maml.xsd'
)
$document.Validate({
    param ($sender, $eventArgs)
    if ($eventArgs.Severity -in 'Error', 'Warning') {
        Write-Host $eventArgs.Message
    }
})
```

The results of the validation are written to the console in the preceding example. The following shows the first message:

```
The element 'details' in namespace 'http://schemas.microsoft.com/maml/dev/command/2004/10' has invalid
child element 'verb' in namespace 'http://schemas.microsoft.com/maml/dev/command/2004/10'. List of
possible elements expected: 'description' in namespace 'http://schemas.microsoft.com/maml/2004/10'.
```

The XML document refers to several schema documents; MAML content is complex. The schema set will only correctly load in .NET Core if `XmlSchemaSet` (`$document.Schemas`) is given an `XmlUrlResolver` to use to process the schema references in the document. This property is mandatory in .NET Core and optional in .NET Framework. The documentation describing the type is available on Microsoft Docs: https://docs.microsoft.com/dotnet/api/system.xml.xmlurlresolver.

The argument for `Validate` is a script block that is executed each time an error is encountered. `Write-Host` is used to print a message to the console. A value cannot be directly returned as the script block is executed in response to an event. A global variable can be used to collect the results for later reference:

```
$document = [Xml]::new()
$document.Load($path)
$document.Schemas.XmlResolver = [System.Xml.XmlUrlResolver]::new()
$document.Schemas.Add(
    'http://schemas.microsoft.com/maml/2004/10',
    'C:\Windows\System32\WindowsPowerShell\v1.0\Schemas\PSMaml\maml.xsd'
)
$validateResult = [System.Collections.Generic.List[object]]::new()
$document.Validate({
    param ($sender, $eventArgs)
    if ($eventArgs.Severity -in 'Error', 'Warning') {
        $validateResult.Add($eventArgs)
    }
})
```

The `$validateResult` list will include each of the failure messages from the validation process. The following shows the first of these:

```
PS> $validateResult[0]
Severity  Exception
--------  ---------
   Error  System.Xml.Schema.XmlSchemaValidationException: The element...
```

The line number and line position information are not available using this technique.

Infer a schema

Normally a schema is created before writing an XML document. It is possible to take an existing XML document and create (or

infer) a schema from the document.

Schema documents can be inferred from an existing XML document using the `XmlSchemaInference` type.

The following document is based on an example used earlier in this chapter:

```
$xml = [Xml]@"
<?xml version="1.0"?>
<cars>
    <car type="Saloon">
        <colour>Green</colour>
    </car>
</cars>
"@
```

`XmlReader` or `XmlNodeReader` is required for the `InferSchema` method. These reader types can be created from the previous XML document we created:

```
$reader = [System.Xml.XmlNodeReader]$xml
```

In the preceding example, `XmlSchema` is created from an existing `XmlDocument` in memory. `XmlReader` can be created using any of the options provided by the `Create` static method of the `XmlReader` class, such as providing a full path to an existing XML file: https://docs.microsoft.com/dotnet/api/system.xml.xmlreader.create?view=netcore-3.1.

`XmlReader` is passed to the `InferSchema` method of the `XmlSchemaInference` type to create a `SchemaSet`.

```
$xmlSchemaInference = [System.Xml.Schema.XmlSchemaInference]::new()
$schemaSet = $xmlSchemaInference.InferSchema($reader)
```

In this example, the `SchemaSet` will contain a single document, which can be written to a stream or `XmlWriter`. `StringWriter` is used to write the schema in the proceeding code; the `ToString` method used at the end retrieves the schema that has been created.

The following shows the complete example with all the preceding steps:

```
$xml = [Xml]@"
<?xml version="1.0"?>
<cars>
    <car type="Saloon">
        <colour>Green</colour>
    </car>
</cars>
"@
$reader = [System.Xml.XmlNodeReader]$xml
$xmlSchemaInference = [System.Xml.Schema.XmlSchemaInference]::new()
$schemaSet = $xmlSchemaInference.InferSchema($reader)
$writer = [System.IO.StringWriter]::new()
$schemaSet.Schemas()[0].Write($writer)
$writer.ToString()
```

The result is an XML schema document describing the structure of the XML document.

```
<?xml version="1.0" encoding="utf-16"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="cars">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="car">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="colour" type="xs:string" />
            </xs:sequence>
            <xs:attribute name="type" type="xs:string" use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

If the XML document contains namespace declarations, a schema will be created for each namespace.

The reader can focus on a smaller part of the document by selecting an element and creating an `XmlNodeReader` at that point in the document. For example, the following snippet creates a schema for the `engine` element only:

```
$reader = [System.Xml.XmlNodeReader]$xml.SelectSingleNode('//engine')
$schemaSet = $xmlSchemaInference.InferSchema($reader)
$writer = [System.IO.StringWriter]::new()
$schemaSet.Schemas()[0].Write($writer)
$writer.ToString()
```

The resultant schema can be modified as appropriate to meet the needs of the document it is intended to validate.

The `System.Xml` namespace contains the classes that are most used in PowerShell. .NET also includes an alternative set of classes to work with XML documents.

System.Xml.Linq

The `System.Xml.Linq` namespace was added with .NET 3.5. This is known as LINQ to XML. **Language Integrated Query** (**LINQ**) is used to describe a query in the same language as the rest of a program. Therefore, interacting with a complex XML document does not require the use of `XPath` queries.

`System.Xml.Linq` is loaded by default in PowerShell 7. Windows PowerShell can make use of `System.Xml.Linq` once the required assembly has been added:

```
Add-Type -AssemblyName System.Xml.Linq
```

This can also be phrased as follows:

```
using assembly System.Xml.Linq
```

As a newer interface, `System.Xml.Linq` tends to be more consistent. The same syntax is used to create a document from scratch that is used to add elements and so on.

Opening documents

The `XDocument` class is used to load or parse a document. XML content may be cast to `XDocument` in the same way that content is cast using the `[Xml]` type accelerator:

```
[System.Xml.Linq.XDocument]$xDocument = @"
<?xml version="1.0"?>
<cars>
    <car type="Saloon">
        <colour>Green</colour>
        <doors>4</doors>
        <transmission>Automatic</transmission>
        <engine>
            <size>2.0</size>
            <cylinders>4</cylinders>
        </engine>
    </car>
</cars>
"@
$xDocument.Save("$pwd\cars.xml")
```

If the content has been saved to a file, the `Load` method may be used with a filename:

```
$xDocument = [System.Xml.Linq.XDocument]::Load("$pwd\cars.xml")
```

Selecting nodes

LINQ to XML uses PowerShell to query the content of XML files. This is achieved by combining the methods that are made available through `XDocument` (or `XContainer` or `XElement`). Methods are available to find attributes and elements, either as immediate children or deeper within a document:

```
$xDocument = [System.Xml.Linq.XDocument]::Load("$pwd\cars.xml")
$xDocument.Descendants('car').
    Where( { $_.Element('colour').Value -eq 'Green' } ).
    Element('engine')
```

As the query a script block encapsulated by the `Where` method is native PowerShell, the comparison operation (`-eq`) is case insensitive. The selection of the element by name is case sensitive.

Although it is not the preferred approach, `XPath` can still be used by calling the `XPathSelectElements` static method, as shown here:

```
[System.Xml.XPath.Extensions]::XPathSelectElements(
    $xDocument,
    '//car[colour="Green"]/engine'
)
```

Creating documents

`System.Xml.Linq` can be used to create a document from scratch, for example:

```
using namespace System.Xml.Linq

$xDocument = [XDocument]::new(
    [XDeclaration]::new('1.0', 'utf-8', 'yes'),
    [XElement]::new('list', @(
        [XAttribute]::new('type', 'numbers'),
        [XElement]::new('name', 1),
        [XElement]::new('name', 2),
        [XElement]::new('name', 3)
    ))
)
```

Using the namespace System.Xml.Linq

The use of the namespace `System.Xml.Linq` is assumed in the remainder of the examples to reduce repetition of the namespace.

Converting the `xDocument` object into a string shows the document without the declaration:

```
PS> $xDocument.ToString()

<list type="numbers">
    <name>1</name>
    <name>2</name>
    <name>3</name>
</list>
```

The `Save` method may be used to write the document to a file:

```
$xDocument.Save("$pwd\test.xml")
```

Reviewing the document shows the declaration:

```
PS> Get-Content test.xml
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<list type="numbers">
    <name>1</name>
    <name>2</name>
    <name>3</name>
</list>
```

Like the types in the `System.Xml` namespace, `System.Xml.Linq` must be told about any namespaces in use.

Working with namespaces

LINQ to XML handles the specification of namespaces by adding an `XNamespace` object to an `XName` object, for example:

```
PS> [XNameSpace]'http://example/cars' + [XName]'engine'

LocalName    Namespace              NamespaceName
---------    ---------              -------------
engine       http://example/cars    http://example/cars
```

As `XNamespace` expects to have `XName` added to it, casting to that type can be skipped, simplifying the expression:

```
[XNamespace]'http://example/cars' + 'engine'
```

A query for an element in a specific namespace will use the following format:

```
using namespace System.Xml.Linq

[XDocument]$xDocument = @"
<?xml version="1.0"?>
<cars xmlns:c="http://example/cars">
    <car type="Saloon">
        <c:colour>Green</c:colour>
        <c:doors>4</c:doors>
        <c:transmission>Automatic</c:transmission>
        <c:engine>
            <size>2.0</size>
            <cylinders>4</cylinders>
        </c:engine>
    </car>
</cars>
"@

$xNScars = [XNameSpace]'http://example/cars'
$xDocument.Descendants('car').ForEach( {
    $_.Element($xNScars + 'engine')
} )
```

Modifying element and attribute values

Modifying an existing node, whether it is an attribute or an element value, can be done by assigning a new value. In the following example, the `category` value is used to correct a typo in the `Fridge` item of the XML document:

```
[XDocument]$xDocument = @"
<?xml version="1.0"?>
<items>
    <item name='Fridge'>
        <category>Appliances</category>
    </item>
    <item name='Cooker'>
        <category>Appliances</category>
    </item>
</items>
"@
$xDocument.Element('items').
    Elements('item').
    Where( { $_.Attribute('name').Value -eq 'Fridge' } ).
    ForEach( { $_.Element('category').Value = 'Appliances' } )
```

Modifying the value of an attribute uses the same syntax:

```
[XDocument]$xDocument = @"
<?xml version="1.0"?>
<list name='letters'>
    <name>1</name>
</list>
"@
$xDocument.Element('list').Attribute('name').Value = 'numbers'
```

If the attribute does not exist, an error will be thrown:

```
PS> $xDocument.Element('list').Attribute('other').Value = 'numbers'
InvalidOperation: The property 'Value' cannot be found on this object. Verify that the property exists
and can be set.
```

The changes made by the preceding assignments may be viewed in the `xDocument` variable:

```
PS> $xDocument.ToString()
<list name="numbers">
  <name>1</name>
</list>
```

New nodes (elements and attributes) may be added to an existing document.

Adding nodes

Nodes can be added to an existing document by using the `Add` methods, which include `Add`, `AddAfterSelf`, `AddBeforeSelf`, and `AddFirst`, for example:

```
[XDocument]$xDocument = @"
<?xml version="1.0"?>
<list type='numbers'>
    <name>1</name>
</list>
"@
$xDocument.Element('list').
    Element('name').
    AddAfterSelf(@(
        [XElement]::new('name', 2),
        [XElement]::new('name', 3),
        [XElement]::new('name', 4)
    ))
```

The different `Add` methods afford a great deal of flexibility over the content of a document; in this case, the new elements appear after the `<name>1</name>` element.

Removing nodes

The `Remove` method of `XElement` or `XAttribute` is used to remove the current node.

In the following example, the first `name` element is removed from the document:

```
[XDocument]$xDocument = @"
<?xml version="1.0"?>
<list type='numbers'>
    <name>1</name>
    <name>2</name>
    <name>3</name>
</list>
"@
$xDocument.Element('list').FirstNode.Remove()
```

Schema validation

LINQ to XML can be used to validate an XML document against a schema file.

The `ISE-help.xml` XML document is validated against its schema in the following example:

```
using namespace System.Xml.Linq

$path = 'C:\Windows\System32\WindowsPowerShell\v1.0\modules\ISE\en-US\ISE-help.xml'
$xDocument = [XDocument]::Load(
    $path,
    [LoadOptions]::SetLineInfo
)

$xmlSchemaSet = [System.Xml.Schema.XmlSchemaSet]::new()
$xmlSchemaSet.XmlResolver = [System.Xml.XmlUrlResolver]::new()
$null = $xmlSchemaSet.Add(
    'http://schemas.microsoft.com/maml/2004/10',
    'C:\Windows\System32\WindowsPowerShell\v1.0\Schemas\PSMaml\maml.xsd'
)
[System.Xml.Schema.Extensions]::Validate(
    $xDocument,
    $xmlSchemaSet,
    {
        param ($sender, $eventArgs)

        if ($eventArgs.Severity -in 'Error', 'Warning') {
            Write-Host $eventArgs.Message
            Write-Host ('  At {0} column {1}' -f
                $sender.LineNumber,
                $sender.LinePosition
            )
        }
    }
)
```

Positional information is made available by loading `XDocument` with the `SetLineInfo` option.

XML parsing and editing is an important feature in PowerShell. XML remains a popular data format.

JSON

JavaScript Object Notation (JSON) is a lightweight format used to store and transport data.

JSON is like XML in some respects. It is intended to be both human- and machine-readable. Like XML, JSON is written in plain text.

JSON is a form of serialization. Data can be converted to and from a string that represents that data. Like a Hashtable, JSON-formatted objects are made up of key-value pairs, for example:

```
{
    "key1": "value1",
    "key2": "value2"
}
```

PowerShell 3 introduced the `ConvertTo-Json` and `ConvertFrom-Json` commands.

> **Newtonsoft JSON is native in PowerShell 7**
>
> PowerShell 7 (and PowerShell 6) use the JSON library from NewtonSoft to serialize and deserialize JSON content.
>
> Advanced JSON serialization and deserialization are available using the classes in the Newtonsoft namespace. The capabilities are documented on the Newtonsoft website: https://www.newtonsoft.com/json/help/html/Introduction.htm.

In PowerShell 7, the `ConvertTo-Json` and `ConvertFrom-Json` commands gain several new parameters. The parameters are explored in the following sections.

ConvertTo-Json

The `ConvertTo-Json` command can be used to convert a PowerShell object (or Hashtable) into JSON:

```
PS> Get-Process -Id $PID |
>>     Select-Object Name, Id, Path |
>>     ConvertTo-Json

{
    "Name":  "pwsh",
    "Id":  3944,
    "Path":  " C:\\Program Files\\PowerShell\\7\\pwsh.exe"
}
```

By default, `ConvertTo-Json` will convert objects into a depth of two. Running the following code will show how the value for `three` is simplified as a string:

```
@{
    one = @{    # 1st iteration (depth 1)
        two = @{    # 2nd iteration (depth 2)
            three = @{
                four = 'value'
            }
        }
    }
} | ConvertTo-Json
```

The `three` property is present, but the value is listed as `System.Collections.Hashtable`, as acquiring the value would need a third iteration. Setting the `-Depth` parameter of `ConvertTo-Json` to `3` allows the properties under the key `three` to convert.

Going too deep

JSON serialization is a recursive operation. The depth may be increased, which is useful when converting a complex object.

Some value types may cause `ConvertTo-Json` to apparently hang. This is caused by the complexity of those value types. Such value types may include circular references.

A `ScriptBlock` object, for example, cannot be efficiently serialized as JSON. The following command takes over 15 seconds to complete and results in a string that's over 50 million characters long:

```
Measure-Command { { 'ScriptBlock' } | ConvertTo-Json -Depth 6 -Compress }
```

Increasing the recursion depth to 7 results in an error as keys (property names) begin to duplicate.

EnumsAsStrings

When `ConvertTo-Json` encounters an enumeration value, it writes that value as a numeric value by default. For example, `DayOfWeek` is written as a numeric value in the following:

```
@{ Today = (Get-Date).DayOfWeek } | ConvertTo-Json
```

Conversely, running `(Get-Date).DayOfWeek` alone will return the name of the day. This change occurs because the `DayOfWeek` value is taken from the `System.DayOfWeek` enumeration.

In PowerShell 7, the switch parameter `EnumsAsStrings` may be used to write the value as a string instead of a number.

```
PS> @{ Today = (Get-Date).DayOfWeek } | ConvertTo-Json -EnumsAsStrings
{
  "Today": "Sunday"
}
```

In Windows PowerShell, the `ConvertTo-Json` command is unable to create an array when a single object is piped. This is where `AsArray` comes in handy.

AsArray

The `ConvertFrom-Json` command is normally used as part of a pipeline. In a pipeline, the command is unable to determine whether the input object was originally an array.

In the following example the input is an array, but only one value is sent to the input pipeline for `ConvertTo.Json`.

```
@(Get-Process -ID $PID | Select-Object Name, ID) | ConvertTo-Json
```

To create a JSON array in Windows PowerShell, the value must be explicitly passed to the `InputObject` parameter:

```
ConvertTo-Json -InputObject @(
    Get-Process -ID $PID | Select-Object Name, ID
)
```

In PowerShell 7, the `AsArray` parameter can be used instead:

```
Get-Process -ID $PID | Select-Object Name, ID | ConvertTo-Json -AsArray
```

PowerShell 7 also offers control over how certain characters should be escaped by using the `-EscapeHandling` parameter.

EscapeHandling

PowerShell 7 includes greater control over escape characters. By default, only JSON control characters are escaped in content. Additional options are included to escape all non-ASCII content (`EscapeNonAscii`), and to escape HTML control characters (`EscapeHtml`).

Support for these new values is provided by the Newtonsoft library:
https://www.newtonsoft.com/json/help/html/T_Newtonsoft_Json_StringEscapeHandling.htm.

For example, the following string contains an accent over one character. The accent is preserved when converting to JSON:

```
PS> @{ String = 'Halló heimur' } | ConvertTo-Json
{
  "String": "Halló heimur"
}
```

If the `EscapeNonAscii` value for the `-EscapeHandling` parameter is used, the resulting string will include a Unicode character sequence instead:

```
PS> @{ String = 'Halló heimur' } |
>>     ConvertTo-Json -EscapeHandling EscapeNonAscii
{
  "String": "Hall\u00f3 heimur"
}
```

This adds flexibility, allowing PowerShell to generate JSON content for a wider variety of other languages and applications.

PowerShell will correctly interpret either format when reading content in with the `ConvertFrom-Json` command.

ConvertFrom-Json

The `ConvertFrom-Json` command is used to turn a JSON document into an object, for example:

```
'{ "Property": "Value" }' | ConvertFrom-Json
```

`ConvertFrom-Json` creates a `PSCustomObject` from the JSON content by default.

JSON understands several different data types, and each of these types is converted into an equivalent .NET type. The following example shows how each different type might be represented:

```
$object = @"
{
    "Decimal": 1.23,
    "String": "string",
    "Int32": 1,
    "Int64": 2147483648,
    "Boolean": true
}
"@ | ConvertFrom-Json
```

Once converted, the result is a custom object as the following shows:

```
PS> $object
Decimal : 1.23
String  : string
Int32   : 1
Int64   : 2147483648
Boolean : True
```

Inspecting individual elements after conversion reflects the type, as demonstrated in the following example:

```
PS> $object.Int64.GetType()
PS> $object.Boolean.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Int64                                    System.ValueType
True     True     Boolean                                  System.ValueType
```

JSON serialization within PowerShell is useful, but it is not perfect. For example, consider the result of converting `Get-Date`:

```
PS> Get-Date | ConvertTo-Json
{
    "value":  "\/Date(1489321529249)\/",
    "DisplayHint":  2,
    "DateTime":  "12 March 2017 12:25:29"
}
```

The value includes a `DisplayHintNoteProperty` and a `DateTimeScriptProperty`, added to the `DateTime` object. These add an extra layer of properties when converting back from JSON:

```
PS> Get-Date | ConvertTo-Json | ConvertFrom-Json

value                  DisplayHint     DateTime
-----                  -----------     --------
12/03/2017 12:27:25              2     12 March 2017 12:27:25
```

The `DateTime` property can be removed using the following code:

```
Get-TypeData System.DateTime | Remove-TypeData
```

Dates without type data

`Get-Date` will appear to return nothing after running the previous command. The date is still present; this is an aesthetic problem only. Without the type data, PowerShell does not know how to display the date, which is ordinarily composed as follows:

```
$date = Get-Date
'{0} {1}' -f $date.ToLongDateString(), $date.ToLongTimeString()
```

`DisplayHint` is added by `Get-Date`, and therefore the command cannot be used when creating a date object for use in a JSON string.

Any extraneous members such as this would have to be tested for invalid members prior to conversion, which makes the solution more of a problem:

```
PS> Get-TypeData System.DateTime | Remove-TypeData
PS> [DateTime]::Now | ConvertTo-Json | ConvertFrom-Json | Select-Object *

Date        : 12/03/2017 00:00:00
Day         : 12
DayOfWeek   : Sunday
DayOfYear   : 71
Hour        : 12
Kind        : Utc
Millisecond : 58
Minute      : 32
Month       : 3
Second      : 41
Ticks       : 636249187610580000
TimeOfDay   : 12:32:41.0580000
Year        : 2017
```

By default, PowerShell converts JSON data into a custom object. In some cases, it may be preferable to read JSON content as a Hashtable.

AsHashtable

By default, `ConvertFrom-Json` creates a `PSCustomObject` from the JSON content. This conversion includes nested objects in the JSON content.

In PowerShell 7, you can use the `AsHashtable` parameter may to create a Hashtable instead of a `PSCustomObject`.

```
$hashtable = @"
{
    "Decimal": 1.23,
    "String": "string",
    "Int32": 1,
    "Int64": 2147483648,
    "Boolean": true
}
"@ | ConvertFrom-Json -AsHashtable
```

If the JSON content includes nested objects, they are also converted into Hashtables:

```
$hashtable = @"
{
    "Key": "Value",
    "Nested": {
        "Key": "NestedValue"
    }
}
"@ | ConvertFrom-Json -AsHashtable
```

In PowerShell, Hashtable keys are not case-sensitive by default when created using `@{}`. Hashtables created by `ConvertFrom-Json` have case-sensitive keys.

NoEnumerate

The `NoEnumerate` parameter is relevant when the root element in a document is an array, and that array contains only one element.

By default, `ConvertFrom-Json` will squash the array, returning a scalar value:

```
PS> $content = @"
>> [
>>     { "Element": 1 }
>> ]
>> "@ | ConvertFrom-Json
PS> $content.GetType()
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     False    PSCustomObject                           System.Object
```

If `NoEnumerate` is used, the resultant type will be an array, `Object[]`.

```
PS> $content = @"
>> [
>>     { "Element": { "Value": 1 } }
>> ]
>> "@ | ConvertFrom-Json -NoEnumerate
PS> $content.GetType()
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array
```

JSON is an incredibly useful format, as it is available in many other languages and used by many other systems. It is an excellent way of drawing data into PowerShell or passing data along to another application or language.

Summary

This chapter took a brief look at working with HTML content and how HTML content is formatted.

Working with XML content is a common requirement. This chapter introduced the structure of XML, along with two different approaches to working with XML.

Finally, JSON serialization was introduced, along with the `ConvertTo-Json` and `ConvertFrom-Json` commands.

Chapter 13, Web Requests and Web Services, explores working with **Representational State Transfer** (**REST**) and **Simple Object Access Protocol** (**SOAP**)-based web services in PowerShell.

13

Web Requests and Web Services

Representational State Transfer (**REST**) and **Simple Object Access Protocol** (**SOAP**) are often used as labels to refer to two different approaches to implementing a web-based **Application Programming Interface** (**API**).

The growth of cloud-based services in recent years has pushed the chances of working with such interfaces from rare to almost certain.

This chapter covers the following topics:

- Web requests
- Working with REST
- Working with SOAP

Before beginning with the main topics of the chapter, there are some technical requirements to consider.

Technical requirements

In addition to PowerShell and PowerShell Core, you'll need Visual Studio 2019 Community Edition or better to follow the SOAP service example.

SOAP interfaces typically use the `New-WebServiceProxy` command in Windows PowerShell. This command is not available in PowerShell 7 as the assembly it depends on is not available. The command is unlikely to be available in PowerShell Core unless it is rewritten.

Web requests, such as getting content from a website or downloading files, are a common activity in PowerShell, and this is the first topic we'll cover.

Web requests

A background in web requests is valuable before delving into interfaces that run over the top of the **Hypertext Transfer Protocol** (**HTTP**).

PowerShell can use `Invoke-WebRequest` to send HTTP requests. For example, the following command will return the response to a `GET` request for the *Hey, Scripting Guy blog*:

```
Invoke-WebRequest https://blogs.technet.microsoft.com/heyscriptingguy/
```

> **Parsing requires Internet Explorer**
>
> In Windows PowerShell, `UseBasicParsing` was an important parameter. Its use was mandatory when working on Core installations of Windows Server as Internet Explorer was not installed. It was also often used to improve the performance of a command where parsing was not actually required.
>
> In PowerShell Core, all requests use basic parsing. The parameter is deprecated and present to support backward compatibility only. The parameter does not affect the output of the command.

A web request has an HTTP method. A request might use SSL, and sometimes may need to work with invalid or self-signed certificates. The web request is the foundation for working with web-based APIs.

HTTP methods

HTTP supports several different methods, including the following:

- `GET`
- `HEAD`
- `POST`
- `PUT`
- `DELETE`
- `CONNECT`
- `OPTIONS`
- `TRACE`
- `PATCH`

These methods are defined in the HTTP 1.1 specification: https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html.

It is common to find that a web server only supports a subset of these. In many cases, supporting too many methods is deemed to be a security risk. The `Invoke-WebRequest` command can be used to verify the list of HTTP methods supported by a site, for example:

```
PS> Invoke-WebRequest www.indented.co.uk -Method OPTIONS |
>>    Select-Object -ExpandProperty Headers

Key            Value
---            -----
Allow          GET, HEAD
```

HTTPS

If a connection to a web service uses HTTPS (HTTP over **Secure Sockets Layer** (**SSL**)), the certificate must be validated before a connection can complete and a request can be completed. If a web service has an invalid certificate, an error will be returned.

The badssl site can be used to test how PowerShell might react to different SSL scenarios: https://badssl.com/.

For example, when attempting to connect to a site with an expired certificate (using `Invoke-WebRequest`), the following message will be displayed in Windows PowerShell:

```
PS> Invoke-WebRequest https://expired.badssl.com/
Invoke-WebRequest : The underlying connection was closed: Could not establish trust relationship for
the SSL/TLS secure channel.
At line:1 char:1
+ Invoke-WebRequest https://expired.badssl.com/
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (System.Net.HttpWebRequest:HttpWebRequest) [Invoke-
WebRequest], WebException
    + FullyQualifiedErrorId :
WebCmdletWebResponseException,Microsoft.PowerShell.Commands.InvokeWebRequestCommand
```

In PowerShell 7, this message changes to `Invoke-WebRequest: The remote certificate is invalid according to the validation procedure.`

In Windows PowerShell, `Invoke-WebRequest` cannot bypass or ignore an invalid certificate on its own (using a parameter). Certificate validation behavior may be changed by adjusting the `CertificatePolicy` on the `ServicePointManager`: https://docs.microsoft.com/dotnet/api/system.net.servicepointmanager?view=netframework-4.8.

The `CertificatePolicy` is a process-specific policy control affecting web requests in .NET made by that process. The policy is applied using the `ServicePointManager`, which manages HTTP connections.

Removing certificate handling changes

PowerShell should be restarted to reset the certificate policies to system defaults. Changes made by `ServicePointManager` apply to the current PowerShell process only and do not persist once PowerShell is closed.

In PowerShell 7, `Invoke-WebRequest` has a parameter that allows certificate errors to be ignored, as the following shows:

```
Invoke-WebRequest https://expired.badssl.com/ -SkipCertificateCheck
```

Chain of trust

Certificates are based on a chain of trust. Authorities are trusted to carry out sufficient checks to prove the identity of the certificate holder. Skipping certificate validation is insecure and should only be used against known hosts that can be trusted.

It may be necessary to ignore an SSL error when making a web request for any number of reasons; for example, the certificate might be self-signed. SSL errors in PowerShell 7 can be bypassed using the preceding parameter. SSL errors in Windows PowerShell are explored in the next section.

Bypassing SSL errors in Windows PowerShell

If a service has an invalid certificate, the best response is to fix the problem. When it is not possible or practical to address the real problem, a workaround can be created.

The approach described here applies to Windows PowerShell only. PowerShell Core does not include the `ICertificatePolicy` type.

This modification applies to the current PowerShell session and will reset to default behavior every time a new PowerShell session is opened.

The certificate policy used by the `ServicePointManager` can be replaced with a customized handler by writing a class (PowerShell, version 5) that replaces the `CheckValidationResult` method:

```
Class AcceptAllPolicy: System.Net.ICertificatePolicy {
    [Boolean] CheckValidationResult(
        [System.Net.ServicePoint] $servicePoint,
        [System.Security.Cryptography.X509Certificates.X509Certificate] $certificate,
        [System.Net.WebRequest] $webRequest,
        [Int32] $problem
    ) {
        return $true
    }
}
[System.Net.ServicePointManager]::CertificatePolicy = [AcceptAllPolicy]::new()
```

Once the policy is in place, certificate errors will be ignored as the previous method returns `true` no matter its state:

```
PS> Invoke-WebRequest "https://expired.badssl.com/"

StatusCode        : 200
StatusDescription : OK
...
```

CertificatePolicy is obsolete

The `CertificatePolicy` property on the `ServicePointManager` is marked as obsolete in Microsoft Docs. This

approach appears to be required when using `Invoke-WebRequest` in PowerShell 5.

Requests made by `System.Net.WebClient` in Windows PowerShell are satisfied by this simpler approach, which trusts all certificates:

```
[System.Net.ServicePointManager]::ServerCertificateValidationCallback = { $true }
```

This approach is not feasible with PowerShell 7. Requests made using `WebClient` may be replaced by either `Invoke-WebRequest` or the `HttpClient`.

Capturing SSL errors

The `ServerCertificateValidationCallback` property of `ServicePointManager` does not work as expected in PowerShell 7. Attempts to assign and use a script block may result in an error being displayed, as shown here, when making a web request:

```
PS> [System.Net.ServicePointManager]::ServerCertificateValidationCallback = { $true }
PS> $webClient = [System.Net.WebClient]::new()
PS> $webClient.DownloadString('https://expired.badssl.com/')
MethodInvocationException: Exception calling "DownloadString" with "1" argument(s): "The SSL
connection could not be established, see inner exception. There is no Runspace available to run
scripts in this thread. You can provide one in the DefaultRunspace property of the
System.Management.Automation.Runspaces.Runspace type. The script block you attempted to invoke was:
$true "
```

The .NET `SslStream` type (`System.Net.Security.SslStream`) offers a potential alternative for capturing detailed certificate validation information. The method used in the following example works in both Windows PowerShell and PowerShell Core.

This example converts certificate validation information using `Export-CliXml`. Assigning the parameters to a global variable is possible, but certain information is discarded when the callback ends, including the elements of the certificate chain. Note that the following example uses three different namespaces. If the example is pasted into the console, then these `using` statements must appear on a single line, each separated by a semi-colon:

```
using namespace System.Security.Cryptography.X509Certificates
using namespace System.Net.Security
using namespace System.Net.Sockets
$remoteCertificateValidationCallback = {
    param (
        [Object]$sender,
        [X509Certificate2]$certificate,
        [X509Chain]$chain,
        [SslPolicyErrors]$sslPolicyErrors
    )
    $psboundparameters | Export-CliXml C:\temp\CertValidation.xml
    # Always indicate SSL negotiation was successful
    $true
}
try {
    [Uri]$uri = 'https://expired.badssl.com/'
    $tcpClient = [TcpClient]::new()
    $tcpClient.Connect($Uri.Host, $Uri.Port)
    $sslStream = [SslStream]::new(
        $tcpClient.GetStream(),
        # leaveInnerStreamOpen: Close the inner stream when complete
        $false,
        $remoteCertificateValidationCallback
    )
    $sslStream.AuthenticateAsClient($Uri.Host)
} catch {
    throw
} finally {
    if ($tcpClient.Connected) {
        $tcpClient.Close()
    }
}
$certValidation = Import-CliXml C:\temp\CertValidation.xml
```

Once the content of the XML file has been loaded, the content may be investigated. For example, the certificate that was exchanged can be viewed:

```
$certValidation.Certificate
```

Or, the response can be used to inspect all the certificates in the key chain:

```
$certValidation.Chain.ChainElements | ForEach-Object Certificate
```

The `ChainStatus` property exposes details of any errors during chain validation:

```
$certValidation.Chain.ChainStatus
```

`ChainStatus` is summarized by the `SslPolicyErrors` property.

The HTTP methods demonstrated can be put to use with **Representational State Transfer** (**REST**).

Working with REST

A REST-compliant web service allows a client to interact with the service using a set of predefined stateless operations. REST is not a protocol; it is an architectural style.

Whether or not an interface is truly REST-compliant is not particularly relevant when the goal is to use one in PowerShell. Interfaces must be used according to any documentation that has been published.

Invoke-RestMethod

The `Invoke-RestMethod` command can execute methods exposed by web services. The name of a method is part of the **Uniform Resource Identifier** (**URI**); it is important not to confuse this with the `Method` parameter. The `Method` parameter is used to describe the HTTP method. By default, `Invoke-RestMethod` uses HTTP `GET`.

Simple requests

The REST API provided by GitHub may be used to list repositories made available by the PowerShell team.

The API entry point, the common URL all REST methods share, is https://api.github.com as documented in the GitHub reference: https://docs.github.com/rest.

When working with REST, documentation is important. The way an interface is used is common, but the way it may respond is not (as this is an architectural style, not a strict protocol).

The specific method being called is documented on a different page of the following reference: https://docs.github.com/rest/reference/repos#list-user-repositories.

The name of the user forms part of the URI; there are no arguments for this method. Therefore, the following command will execute the method and return detailed information about the repositories owned by the PowerShell user (or organization):

```
Invoke-RestMethod -Uri https://api.github.com/users/powershell/repos
```

Windows PowerShell is likely to throw an error relating to SSL/TLS when running this command. This is because the site uses TLS 1.2 whereas, by default, `Invoke-RestMethod` reaches as far as TLS 1.0. PowerShell 7 users should not experience this problem.

This Windows PowerShell problem can be fixed by tweaking the `SecurityProtocol` property of `ServicePointManager` as follows:

```
using namespace System.Net
[ServicePointManager]::SecurityProtocol = [ServicePointManager]::SecurityProtocol -bor 'Tls12'
```

The bitwise `-bor` operator is used to add TLS 1.2 to the default list defined by a combination of the .NET version and the Windows version. That typically includes `Ssl3` and `Tls`. TLS 1.1 (`Tls11`) may be added in a similar manner if required. The change applies to the current PowerShell process and does not persist.

> **All examples use TLS 1.2**
>
> This setting is required for the examples that follow when running Windows PowerShell.
>
> Older versions of Windows may require a patch from Windows Update to gain support for TLS 1.2.

Requests with arguments

The search code method of the GitHub REST API is used to demonstrate how arguments can be passed to a REST method.

The documentation for the method is found in the following API reference: https://docs.github.com/rest/reference/search#search-code.

The following example uses the search code method by building a query string and appending that to the end of the URL. The search looks for occurrences of the `Get-Content` term in PowerShell language files in the PowerShell repository. The search term is therefore as follows; the search string here should not be confused with a PowerShell command:

```
Get-Content language:powershell repo:powershell/powershell
```

Get-Content is not the PowerShell command Get-Content

PowerShell has a `Get-Content` command. The `Get-Content` term used in the previous search string should not be confused with the PowerShell command.

Converting the example from the documentation for the search method, the URL required is as follows. Spaces may be replaced by `+` when encoding the URL: https://api.github.com/search/code?q=Get-Content+language:powershell+repo:powershell/powershell.

In Windows PowerShell, which can use the `HttpUtility` type within the `System.Web` assembly, the task of encoding the URL can be simplified:

```
using assembly System.Web
$queryString = [System.Web.HttpUtility]::ParseQueryString('')
$queryString.Add(
    'q',
    'Get-Content language:powershell repo:powershell/powershell'
)
$uri = 'https://api.github.com/search/code?{0}' -f $queryString
Invoke-RestMethod $uri
```

The result is a custom object that includes the search results:

```
PS> Invoke-RestMethod $uri
total_count incomplete_results items
----------- ------------------ -----
         80              False {@{name=Get-Content.Tests....
```

Running `$queryString.ToString()` shows that the colon character has been replaced by `%3`, and the forward-slash in the repository name has been replaced by `%2`. The `%3` and `%2` are HTTP encodings of the colon and forward-slash characters.

The arguments for the search do not necessarily have to be passed as a query string. Instead, a body for the request may be set, as shown here:

```
Invoke-RestMethod -Uri https://api.github.com/search/code -Body @{
    q = 'Get-Content language:powershell repo:powershell/powershell'
}
```

`Invoke-RestMethod` converts the body (a Hashtable) to JSON and handles any encoding required. The result of the search is the same whether the body or a query string is used.

In both cases, details of the files found are held within the `items` property of the response. The following example shows the file name and path:

```
$params = @{
    Uri = 'https://api.github.com/search/code'
    Body = @{
        q = 'Get-Content language:powershell repo:powershell/powershell'
    }
}
Invoke-RestMethod @params |
    Select-Object -ExpandProperty items |
    Select-Object name, path
```

This pattern, where the actual results are nested under a property in the response, is frequently seen with REST interfaces. Exploration is often required.

It is critical to note that REST interfaces are case-sensitive; using a parameter named `Q` would result in an error message, as shown here:

```
PS> Invoke-RestMethod -Uri https://api.github.com/search/code -Body @{
>>     Q = 'Get-Content language:powershell repo:powershell/powershell'
>> }
Invoke-RestMethod: {"message":"Validation Failed","errors":
[{"resource":"Search","field":"q","code":"missing"}],"documentation_url":"https://developer.github.com
/v3/search"}
```

The GitHub API returns an easily understood error message in this case. This will not be true of all REST APIs; it is common to see a generic error returned by an API. An API may return a simple HTTP 400 error and leave it to the user or developer to figure out what went wrong.

Working with paging

Many REST interfaces will return large result sets from searches in pages, a subset of the results. The techniques used to retrieve each subsequent page can vary from one API to another. This section explores how those pages can be retrieved from the web service.

The GitHub API exposes the link to the next page in the HTTP header. This is consistent with RFC 5988 (https://tools.ietf.org/html/rfc5988#page-6).

In PowerShell 7, it is easy to retrieve and view the header when using `Invoke-RestMethod`:

```
$params = @{
    Uri                     = 'https://api.github.com/search/issues'
    Body                    = @{
       q = 'documentation state:closed repo:powershell/powershell'
    }
    ResponseHeadersVariable = 'httpHeader'
}
Invoke-RestMethod @params | Select-Object -ExpandProperty items
```

Once run, the `link` field of the header may be inspected via the `httpHeader` variable:

```
PS> $httpHeader['link']
 <https://api.github.com/search/issues?
q=documentation+state%3Aclosed+repo%3Apowershell%2Fpowershell&page=2>; rel="next",
 <https://api.github.com/search/issues?
q=documentation+state%3Aclosed+repo%3Apowershell%2Fpowershell&page=34>; rel="last"
```

PowerShell 7 can also automatically follow this link by using the `FollowRelLink` parameter. This might be used in conjunction with the `MaximumFollowRelLink` parameter to ensure a request stays within any rate limiting imposed by the web service. See https://docs.github.com/en/rest/overview/resources-in-the-rest-api#rate-limiting for the GitHub API. The following request searches for closed documentation issues, following the paging link twice:

```
$params = @{
    Uri                  = 'https://api.github.com/search/issues'
    Body                 = @{
       q = 'documentation state:closed repo:powershell/powershell'
    }
    FollowRelLink        = $true
    MaximumFollowRelLink = 2
}
Invoke-RestMethod @params | Select-Object -ExpandProperty items
```

Windows PowerShell, unfortunately, cannot automatically follow this link. Nor does the `Invoke-RestMethod` command expose the header from the response. When working with complex REST interfaces in Windows PowerShell, it is often necessary to fall back to the `Invoke-WebRequest` or even `HttpWebRequest` classes.

The example that follows uses `Invoke-WebRequest` in Windows PowerShell to follow the next link in a similar manner to

`Invoke-RestMethod` in PowerShell 7:

```
# Used to limit the number of times "next" is followed
$followLimit = 2
# The initial set of parameters, describes the search
$params = @{
    Uri = 'https://api.github.com/search/issues'
    # PowerShell will convert this to JSON
    Body = @{
        q = 'documentation state:closed repo:powershell/powershell'
    }
    ContentType = 'application/json'
}
# Just a counter, works in conjunction with followLimit.
$followed = 0
do {
    # Get the next response
    $response = Invoke-WebRequest @params
    # Convert and leave the results as output
    $response.Content |
        ConvertFrom-Json |
        Select-Object -ExpandProperty items
    # Retrieve the links from the header and find the next URL
    if ($response.Headers['link'] -match '<([^>]+?)>;\s*rel="next"') {
        $next = $matches[1]
    } else {
        $next = $null
    }
    # Parameters which will be used to get the next page
    $params = @{
        Uri = $next
    }
    # Increment the followed counter
    $followed++
} until (-not $next -or $followed -ge $followLimit)
```

Because of the flexible nature of REST, implementations of page linking may vary. For example, links may appear in the body of a response instead of the header. Exploration is a requirement when working around a web API.

Working with authentication

There are many authentication systems that can be used when working with web services.

For services that expect to use the current user account to authenticate, the `UseDefaultCredential` parameter may be used to pass authentication tokens without explicitly passing a username and password. A service integrated into an Active Directory domain, expecting to use Kerberos authentication, is an example of such a service.

REST interfaces written to provide automation access tend to offer reasonably simple approaches to automation, often including basic authentication.

GitHub offers several different authentication methods, including basic and OAuth. These are shown here when attempting to request the email addresses for a user that requires authentication.

> **GitHub user basic auth is deprecated**
>
> GitHub removed basic authentication entirely in November 2020. The following basic authentication example will fail unless a personal access token is used as the password.
>
> https://developer.github.com/changes/2020-02-14-deprecating-password-auth/

Using basic authentication

Basic authentication with a username and password is the simplest method available. For the GitHub API, the password must be a personal access token, basic authentication with a username and password was discontinued in November 2020.

You can generate personal access tokens by visiting account settings, then developer settings. Once generated, the personal access token cannot be viewed. The personal access token is used in place of a password:

```
$params = @{
    Uri        = 'https://api.github.com/user/emails'
    Credential = Get-Credential
}
Invoke-RestMethod @params
```

In PowerShell 7, the `Authentication` parameter should be added:

```
$params = @{
    Uri            = 'https://api.github.com/user/emails'
    Credential     = Get-Credential
    Authentication = 'Basic'
}
Invoke-RestMethod @params
```

GitHub provides documentation showing how to add the second authentication factor, although it is not clear how SMS tokens can be requested: https://docs.github.com/rest/overview/other-authentication-methods.

OAuth is an alternative to using a personal access token.

OAuth

OAuth is offered by a wide variety of web services. The details of this process will vary slightly between different APIs. The GitHub documentation describes the process that must be followed: https://docs.github.com/developers/apps/authorizing-oauth-apps#web-application-flow.

Implementing OAuth requires a web browser, or a web browser and a web server. As the browser will likely need to run JavaScript, this cannot be done using `Invoke-WebRequest` alone.

Creating an application

Before starting with code, an application must be registered with GitHub. This is done by visiting **Settings**, and then **Developer settings**, and finally **OAuth Apps**.

A new OAuth app must be created to acquire a `clientId` and `clientSecret`. Creating the application requires a homepage URL and an authorization callback URL. Both should be set to `http://localhost:40000` for this example. This URL is used to acquire the authorization code.

The values from the web page will fill the following variables, the client secret must be generated by clicking **Generate a new client secret**. The secret must be regenerated if lost:

Generate a new client secret in screen text style.

```
$clientId = 'FromGitHub'
$clientSecret = 'FromGitHub'
```

Getting an authorization code

Once an application is registered, an authorization code is required. Obtaining the authorization code gives the end user the opportunity to grant the application access to a GitHub account. If the user is not currently logged in to GitHub, it will also prompt them to log on.

A URL must be created that will prompt for authorization:

```
$authorize = 'https://github.com/login/oauth/authorize?client_id={0}&scope={1}' -f @(
    $clientId
    'user:email'
)
```

The `'user:email'` scope describes the rights the application would like to have. The web API guide contains a list of possible scopes: https://docs.github.com/developers/apps/scopes-for-oauth-apps.

OAuth browser issues

WPF and Windows Forms both include browser controls that can be used. However, both are based on Internet Explorer, which is not supported by GitHub. At this point, a choice must be made. Either a browser must be controlled, which is challenging, or an HTTP listener must be created to receive the response after OAuth completes. An HTTP listener is the easier path as this can leverage any browser installed on the current computer.

Implementing an HTTP listener

Implementing the web server has two advantages:

- Implementing a web server does not need additional libraries
- The web server can potentially be used on platforms other than Windows

The `HttpListener` is configured with the callback URL as a prefix. The prefix must end with a forward slash. The operating system gets to choose which browser should be used to complete the request:

```
$httpListener = [System.Net.HttpListener]::new()
$httpListener.Prefixes.Add('http://localhost:40000/')
$httpListener.Start()
$clientId = Read-Host 'Enter the client-id'
$authorizeUrl = 'https://github.com/login/oauth/authorize?client_id={0}&scope={1}' -f @(
    $clientId
    'user:email'
)
# Let the operating system choose the browser to use for this request
Start-Process -FilePath $authorizeUrl
$context = $httpListener.GetContext()
$buffer = [Byte[]][Char[]]"<html><body>OAuth complete! Please return to PowerShell!</body></html>"
$context.Response.OutputStream.Write(
    $buffer,
    0,
    $buffer.Count
)
$context.Response.OutputStream.Close()
$httpListener.Stop()
$authorizationCode = $context.Request.QueryString['code']
```

In either case, the result of the process is code held in the `$authorizationCode` variable. This code can be used to request an access token.

Requesting an access token

The next step is to create an access token. The access token is valid for a limited time.

The `clientSecret` is sent with this request; if this were an application that was given to others, keeping the secret would be a challenge to overcome:

```
$params = @{
    Uri = 'https://github.com/login/oauth/access_token'
    Method = 'POST'
    Body = @{
        client_id     = $clientId
        client_secret = $clientSecret
        code          = $authorizationCode
    }
}
$response = Invoke-RestMethod @params
$token = [System.Web.HttpUtility]::ParseQueryString($response)['access_token']
```

The previous request used the HTTP method `POST`. The HTTP method, which should be used with a REST method, is documented for an interface in the Developer Guides.

Each of the requests that follow will use the access token from the previous request. The access token is placed in an HTTP header field named `Authorization`.

Using a token

We can call methods that require authentication by adding a token to the HTTP header. The format of the `Authorization` header field is as follows:

```
Authorization: token OAUTH-TOKEN
```

`OAUTH-TOKEN` is replaced, and the `Authorization` head is constructed as follows:

```
$headers = @{
    Authorization = 'token {0}' -f $token
}
```

The token can be used in subsequent requests for the extent of its lifetime:

```
$headers = @{
    Authorization = 'token {0}' -f $token
}
Invoke-RestMethod 'https://api.github.com/user/emails' -Headers $headers
```

Each REST API for each different system or service tends to take a slightly different approach to authentication, authorization, calling methods, and even details like paging. However, despite the differences there, the lessons learned using one API are still useful when attempting to write code for another.

REST is an extremely popular style these days. Before REST became prominent, services built using SOAP were common.

Working with SOAP

Unlike REST, which is an architectural style, SOAP is a protocol. It is perhaps reasonable to compare working with SOAP to importing a .NET assembly (DLL) to work with the types inside. As a result, a SOAP client is much more strongly tied to a server than is the case with a REST interface.

SOAP uses XML to exchange information between the client and server.

Finding a SOAP service

SOAP-based web APIs have become quite rare, less popular by far than REST. The examples in this section are based on a simple SOAP service written for this book.

The service is available on GitHub as a Visual Studio solution. The solution is also available in the GitHub repository containing the code examples used in this chapter: https://github.com/indented-automation/SimpleSOAP.

The solution should be downloaded, opened in Visual Studio (2015 or 2017, Community Edition or better), and debugging should be started by pressing F5. A browser page will be opened, which will show the port number the service is operating on. A `403` error may be displayed; this can be ignored.

> **Localhost and a port**
>
> Throughout this section, localhost and a port are used to connect to the web service. The port is set by Visual Studio when debugging the simple SOAP web service and must be updated to use these examples.

This service is not well-designed; it has been contrived to expose similar patterns in its method calls to those seen in real SOAP services.

A `ReadMe` file accompanies the project. Common problems running the project will be noted there.

The discovery-based approaches explored in this section should be applicable to any SOAP-based service.

SOAP in Windows PowerShell

The `New-WebServiceProxy` examples that follow apply to Windows PowerShell only. The `New-WebServiceProxy` command is not available in PowerShell 7.

New-WebServiceProxy

The `New-WebServiceProxy` command is used to connect to a SOAP web service. This can be a service endpoint, such as a .NET `service.asmx` URL, or a WSDL document.

The web service will include methods and may also include other object types and enumerations.

The command accesses a service anonymously by default. If the current user should be passed on, the `UseDefaultCredential` parameter should be used. If explicit credentials are required, the `Credential` parameter can be used.

By default, `New-WebServiceProxy` creates a dynamic namespace. This is as follows:

```
PS> $params = @{
>>    Uri = 'http://localhost:62369/Service.asmx'
>> }
PS> $service = New-WebServiceProxy @params
PS> $service.GetType().Namespace
Microsoft.PowerShell.Commands.NewWebserviceProxy.AutogeneratedTypes.WebServiceProxy4__localhost_62369_
Service_asmx
```

The dynamic namespace is useful as it avoids problems when multiple connections are made to the same service in the same session.

To simplify exploring the web service, a fixed namespace might be defined:

```
$params = @{
    Uri       = 'http://localhost:62369/Service.asmx'
    Namespace = 'SOAP'
}
$service = New-WebServiceProxy @params
```

The `$service` object returned by `New-WebServiceProxy` describes the URL used to connect, the timeout, the HTTP user agent, and so on. The object is also the starting point for exploring the interface; it is used to expose web services' methods.

Methods

The methods available may be viewed in several ways. The URL used can be visited in a browser or `Get-Member` may be used. A subset of the output from `Get-Member` follows:

```
PS> $service | Get-Member
Name                MemberType  Definition
----                ----------  ----------
GetElement          Method      SOAP.Element GetElement(string Name)
GetAtomicMass       Method      string GetAtomicMass(string Name)
GetAtomicNumber     Method      int GetAtomicNumber(string Name)
GetElements         Method      SOAP.Element[] GetElements()
GetElementsByGroup  Method      SOAP.Element[] GetElementsByGroup(SOAP.Group group)
GetElementSymbol    Method      string GetElementSymbol(string Name)
SearchElements      Method      SOAP.Element[] SearchElements(SOAP.SearchCondition[]
searchConditions)
```

The preceding `GetElements` method requires no arguments and may be called immediately, as shown here:

```
PS> $service.GetElements() | Select-Object -First 5 | Format-Table
AtomicNumber    Symbol    Name         AtomicMass     Group
------------    ------    ----         ----------     -----
           1    H         Hydrogen     1.00794(4)     Nonmetal
           2    He        Helium       4.002602(2)    NobleGas
           3    Li        Lithium      6.941(2)       AlkaliMetal
           4    Be        Beryllium    9.012182(3)    AlkalineEarthMetal
           5    B         Boron        10.811(7)      Metalloid
```

Methods requiring string or numeric arguments may be similarly easy to call, although the value the method requires is often open to interpretation. In this case, the `Name` argument may be either an element name or an element symbol:

```
PS> $service.GetAtomicNumber('oxygen')
8
PS> $service.GetAtomicMass('H')
1.00794(4)
```

Whether the web service is SOAP or REST, using the service effectively is dependent on being able to locate the service documentation.

Methods and enumerations

The `GetElementsByGroup` method shown by `Get-Member` requires an argument of type `SOAP.Group` as the following shows:

```
PS> $service | Get-Member -Name GetElementsByGroup
Name               MemberType Definition
----               ---------- ----------
GetElementsByGroup Method     SOAP.Element[] GetElementsByGroup(SOAP.Group…
```

`SOAP.Group` is an enumeration, as indicated by the `BaseType`:

```
PS> [SOAP.Group]
IsPublic    IsSerial    Name     BaseType
--------    --------    ----     --------
True        True        Group    System.Enum
```

The values of the enumeration may be shown by running the `GetEnumValues` method:

```
PS> [SOAP.Group].GetEnumValues()
Actinoid
AlkaliMetal
AlkalineEarthMetal
Halogen
Lanthanoid
Metal
Metalloid
NobleGas
Nonmetal
PostTransitionMetal
TransitionMetal
```

PowerShell will help cast to enumeration values; a string value is sufficient to satisfy the method:

```
PS> $service.GetElementsByGroup('Nonmetal') | Format-Table
AtomicNumber    Symbol    Name          AtomicMass      Group
------------    ------    ----          ----------      -----
           1    H         Hydrogen      1.00794(4)      Nonmetal
           6    C         Carbon        12.0107(8)      Nonmetal
           7    N         Nitrogen      14.0067(2)      Nonmetal
           8    O         Oxygen        15.9994(3)      Nonmetal
          15    P         Phosphorus    30.973762(2)    Nonmetal
          16    S         Sulfur        32.065(5)       Nonmetal
          34    Se        Selenium      78.96(3)        Nonmetal
```

If the real value of the enumeration must be used, it may be referenced as a static property of the enumeration:

```
$service.GetElementsByGroup([SOAP.Group]::Nonmetal) | Format-Table
```

It is relatively common for a method to require an instance of an object provided by the SOAP interface.

Methods and SOAP objects

When working with SOAP interfaces, it is common to encounter methods that need instances of objects presented by the SOAP service. The `SearchElements` method is an example of this type.

The `SearchElements` method requires an array of `SOAP.SearchCondition` as an argument. This is shown in the following by accessing the definition of the method:

```
PS> $service.SearchElements
OverloadDefinitions
-------------------
SOAP.Element[] SearchElements(SOAP.SearchCondition[] searchConditions)
```

An instance of `SearchCondition` may be created as follows:

```
$searchCondition = [SOAP.SearchCondition]::new()
```

Exploring the object with `Get-Member` shows that the operator property is another type from the SOAP service. This is an enumeration, as shown here:

```
PS> [SOAP.ComparisonOperator]
IsPublic    IsSerial    Name                  BaseType
--------    --------    ----                  --------
True        True        ComparisonOperator    System.Enum
```

A set of search conditions may be constructed and passed to the method:

```
$searchConditions = @(
    [SOAP.SearchCondition]@{
        PropertyName = 'AtomicNumber'
        Operator     = 'ge'
        Value        = 1
    }
    [SOAP.SearchCondition]@{
        PropertyName = 'AtomicNumber'
        Operator     = 'lt'
        Value        = 6
    }
)
$service.SearchElements($searchConditions)
```

Overlapping services

When testing a SOAP interface, it is easy to get into a situation where `New-WebServiceProxy` has been called several times against the same web service. This can be problematic if using the `Namespace` parameter.

Consider the following example, which uses two instances of the web service:

```
$params = @{
    Uri = 'http://localhost:62369/Service.asmx'
    Namespace = 'SOAP'
}
# Original version
$service = New-WebServiceProxy @params
# New version
$service = New-WebServiceProxy @params
$searchConditions = @(
    [SOAP.SearchCondition]@{
        PropertyName = 'Symbol'
        Operator     = 'eq'
        Value        = 'H'
    }
)
```

In theory, there is nothing wrong with this example. In practice, the `SOAP.SearchCondition` object is created based on the original version of the service created using `New-WebServiceProxy`. The method is, on the other hand, executing against the newer version.

As the method being called and the type being used are in different assemblies, an error is shown; this is repeated in the following:

```
PS> $service.SearchElements($searchConditions)
Cannot convert argument "searchConditions", with value: "System.Object[]", for "SearchElements" to
type
"SOAP.SearchCondition[]": "Cannot convert the "SOAP.SearchCondition" value of type
"SOAP.SearchCondition" to type
"SOAP.SearchCondition"."
At line:1 char:1
+ $service.SearchElements($searchConditions)
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 + CategoryInfo : NotSpecified: (:) [], MethodException
 + FullyQualifiedErrorId : MethodArgumentConversionInvalidCastArgument
```

It is still possible to access the second version of `SearchCondition` by searching for the type, then creating an instance of that:

```
$searchCondition = ($service.GetType().Module.GetTypes() |
    Where-Object Name -eq 'SearchCondition')::new()
$searchCondition.PropertyName = 'Symbol'
$searchCondition.Operator = 'eq'
$searchCondition.Value = 'H'
$searchConditions = @($searchCondition)
$service.SearchElements($searchConditions)
```

However, it is generally better to avoid the problem by allowing `New-WebServiceProxy` to use a dynamic namespace. At which point, an instance of the `SearchCondition` may be created, as the following shows:

```
('{0}.SearchCondition' -f $service.GetType().Namespace -as [Type])::new()
```

SOAP in PowerShell 7

Windows PowerShell can use the `New-WebServiceProxy` command. In PowerShell 7, requests can be manually created and sent using `Invoke-WebRequest`.

Getting the WSDL document

The **Web Services Description Language** (**WSDL**) document for the web service contains details of the methods and enumerations it contains. The document can be requested as follows:

```
$params = @{
    Uri = 'http://localhost:62369/Service.asmx?wsdl'
}
[Xml]$wsdl = Invoke-WebRequest @params | Select-Object -ExpandProperty Content
```

The document shows the methods that can be executed and the arguments for those methods. Obtaining a list requires a bit of correlation.

Discovering methods and enumerations

You can use the WSDL document to discover the methods available from the SOAP service. The default view of the document presents a list of methods. Clicking on a method will show an example of the expected header and body values.

It is possible to retrieve the methods in PowerShell dynamically as well. Two SOAP versions are presented by the service. SOAP 1.2 is used in the following example, although both will show the same information in this case:

```
$xmlNamespaceManager = [System.Xml.XmlNamespaceManager]::new($wsdl.NameTable)
# Load everything that looks like a namespace
$wsdl.definitions.PSObject.Properties |
    Where-Object Value -match '^http' |
    ForEach-Object {
        $xmlNamespaceManager.AddNamespace(
            $_.Name,
            $_.Value
        )
    }
$wsdl.SelectNodes(
    '/*/wsdl:binding[@name="ServiceSoap12"]/wsdl:operation',
    $xmlNamespaceManager
).ForEach{
    [PSCustomObject]@{
        Name       = $_.name
        Inputs     = $wsdl.SelectNodes(
            ('//s:element[@name="{0}"]/*/s:sequence/*' -f $_.name),
            $xmlNamespaceManager
        ).ForEach{
            [PSCustomObject]@{
                ParameterName = $_.name
                ParameterType = $_.type -replace '.+:'
            }
        }
        Outputs    = $wsdl.SelectNodes(
            ('//s:element[@name="{0}Response"]/*/*/s:element/@type' -f
                $_.name),
            $xmlNamespaceManager
        ).'#text' -replace '.+:'
        SoapAction = $_.operation.soapAction
    }
}
```

The preceding script shows a rough list of the parameters required and value types returned for each of the methods. For example, the `GetElement` method expects a string argument and will return an `Element` object:

```
Name         Inputs                                         Outputs
----         ------                                         -------
GetElement   {@{ParameterName=Name; ParameterType=string}}  Element
```

Enumeration values are also exposed in the WSDL. Continuing from the previous example, they may be retrieved using the following:

```
$wsdl.SelectNodes(
    '/*/*/*/s:simpleType[s:restriction/s:enumeration]',
    $xmlNamespaceManager
).ForEach{
    [PSCustomObject]@{
        Name   = $_.name
        Values = $_.restriction.enumeration.value
    }
}
```

The `Group` and `ComparisonOperator` enumerations will be displayed.

Running methods

You can use `Invoke-WebRequest` to execute methods by providing a SOAP envelope in the body of the request. The response is an XML document that includes the results of the method. The envelope includes the web service URI, http://tempuri.org:

```
$params = @{
    Uri         = 'http://localhost:62369/Service.asmx'
    ContentType = 'text/xml'
    Method      = 'POST'
    Header      = @{
        SOAPAction = 'http://tempuri.org/GetElements'
    }
    Body        = '
        <soapenv:Envelope
                xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
            <soapenv:Body>
                <GetElements />
            </soapenv:Body>
        </soapenv:Envelope>
    '
}
$webResponse = Invoke-WebRequest @params
$xmlResponse = [Xml]$webResponse.Content
$body = $xmlResponse.Envelope.Body
$body.GetElementsResponse.GetElementsResult.Element
```

As the preceding code shows, the response content is specific to the method that was executed.

If a method requires arguments, these must be passed in the body of the request. In the following example, the argument is a single string:

```
$params = @{
    Uri         = 'http://localhost:62369/Service.asmx'
    ContentType = 'text/xml'
    Method      = 'POST'
    Header      = @{
        SOAPAction = 'http://tempuri.org/GetElement'
    }
    Body        = '
        <soapenv:Envelope
                xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                xmlns="http://tempuri.org/">
            <soapenv:Body>
                <GetElement>
                    <Name>Oxygen</Name>
                </GetElement>
            </soapenv:Body>
        </soapenv:Envelope>
    '
}
$webResponse = Invoke-WebRequest @params
$xmlResponse = [Xml]$webResponse.Content
$body = $xmlResponse.Envelope.Body
```

The body shows the object returned by the method:

```
PS> $body.GetElementResponse.GetElementResult
AtomicNumber : 8
Symbol       : O
Name         : Oxygen
AtomicMass   : 15.9994(3)
Group        : Nonmetal
Requests requiring an enumeration value, such as Get
```

The name of the argument used above correlates with the name and value shown in the WSDL:

```
<s:element minOccurs="0" maxOccurs="1" name="Name" type="s:string"/>
```

The preceding method expects a string. If an enumeration value is required, it can be described as a string in the XML envelope.

More complex types can be built based on following the expected structure of the arguments, or following the examples provided when browsing the `Service.asmx` file. The following example includes two `SearchCondition` objects:

```
$params = @{
    Uri         = 'http://localhost:62369/Service.asmx'
    ContentType = 'text/xml'
    Method      = 'POST'
    Body        = '
        <soapenv:Envelope
                xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                xmlns="http://tempuri.org/">
            <soapenv:Body>
                <SearchElements>
                    <searchConditions>
                        <SearchCondition>
                            <PropertyName>AtomicNumber</PropertyName>
                            <Value>1</Value>
                            <Operator>ge</Operator>
                        </SearchCondition>
                        <SearchCondition>
                            <PropertyName>AtomicNumber</PropertyName>
                            <Value>6</Value>
                            <Operator>lt</Operator>
                        </SearchCondition>
                    </searchConditions>
                </SearchElements>
            </soapenv:Body>
        </soapenv:Envelope>
    '
}
$webResponse = Invoke-WebRequest @params
$xmlResponse = [Xml]$webResponse.Content
$body = $xmlResponse.Envelope.Body
$body.SearchElementsResponse.SearchElementsResult.Element
```

`New-WebServiceProxy` in Windows PowerShell took away some of the difficulty of defining the SOAP envelope, but in most cases, it is not difficult to create.

Summary

This chapter explored the use of `Invoke-WebRequest` and how to work with and debug SSL negotiation problems.

Working with REST explored simple method calls, authentication, and OAuth negotiation, before exploring REST methods that require authenticated sessions.

SOAP is hard to find these days; a sample project was used to show how the capabilities of a SOAP service might be discovered and used.

Chapter 14, Remoting and Remote Management, explores remoting and remote management.

14

Remoting and Remote Management

Windows remoting came to PowerShell with the release of version 2.0. Windows remoting is a powerful feature that allows administrators to move away from RPC-based remote access.

This chapter covers the following topics:

- WS-Management
- PSSessions
- Remoting on Linux
- Remoting over SSH
- The double-hop problem
- CIM sessions
- Just Enough Administration

Before beginning with the main topics of the chapter, there are some technical requirements to consider.

Technical requirements

This chapter makes use of a remote Windows system named PSTest, which runs Windows 10, Windows PowerShell 5.1, and PowerShell 7.

Remoting between Windows and Linux is demonstrated using a system that runs CentOS 7, PowerShell 7, and the **PowerShell Remoting Protocol** (**PSRP**) package.

Windows makes use of WSMan, or WS-Management, via PSRP.

WS-Management

Windows remoting uses WS-Management as its communication protocol. Support for WS-Management and remoting was introduced with PowerShell 2.0. WS-Management uses **Simple Object Access Protocol** (**SOAP**) to pass information between the client and the server.

PowerShell Remoting Protocol, PSRP, uses WS-Management as a means of communicating with a remote PowerShell instance.

Enabling remoting

Before remoting can be used on a desktop operating system, it must be enabled. In a domain environment, remoting can be enabled using a group policy:

- **Policy name**: Allow remote server management through WinRM
- **Path**: `Computer configuration\Administrative Templates\Windows Components\Windows Remote Management (WinRM)\WinRM Service`

If remoting is enabled using a group policy, a firewall rule should be created to allow access to the service:

- **Policy name**: Define inbound port exceptions
- **Path**: `Computer Configuration\Administrative Templates\Network\Network Connections\Windows Firewall\Domain Profile`

- **Port exception example**: `5985:TCP:*:enabled:WSMan`

You can enable Windows remoting on a per-machine basis using the `Enable-PSRemoting` command.

Remoting may be disabled in PowerShell using `Disable-PSRemoting`. Disabling remoting will show the following warning:

```
PS> Disable-PSRemoting
WARNING: Disabling the session configurations does not undo all the changes made by the Enable-
PSRemoting or Enable-PSSessionConfiguration cmdlet. You might have to manually undo the changes by
following these steps:
1. Stop and disable the WinRM service.
2. Delete the listener that accepts requests on any IP address.
3. Disable the firewall exceptions for WS-Management communications.
4.Restore the value of the LocalAccountTokenFilterPolicy to 0, which restricts remote access to
members of the Administrators group on the computer.
```

If `Enable-PSRemoting` is run in the PowerShell 7 console, additional session configurations will be created that allow a choice of either Windows PowerShell (the default) or PowerShell 7 when creating a remote session. Accessing PowerShell 7 sessions is explored later in this chapter.

Get-WSManInstance

`Get-WSManInstance` provides access to instances of resources at a lower level than commands such as `Get-CimInstance`.

For example, `Get-WSManInstance` can be used to get the `Win32_OperatingSystem` WMI class:

```
Get-WSManInstance -ResourceUri wmicimv2/win32_operatingsystem
```

The response is an `XmlElement` that PowerShell presents as an object with properties for each child element.

`Get-WSManInstance` has been superseded by `Get-CimInstance`, which was introduced in PowerShell 3.0.

The `Get-WSManInstance` command is only supported on Windows installations of PowerShell.

The WSMan drive

The content of the `WSMan` drive is accessible when PowerShell is running as the administrator. The drive can be used to view and change the configuration of remoting.

For example, the provider can be used to update settings such as `MaxEnvelopeSize`, which affects the maximum permissible size of SOAP messages sent and received by `WSMan`:

```
Set-Item WSMan:\localhost\MaxEnvelopeSizekb 8KB
```

The `MaxEnvelopeSize` property is defined in the WSMan protocol extension's specification:

https://docs.microsoft.com/openspecs/windows_protocols/ms-wsman/8a6b1967-ff8e-4756-9a3b-890b4b439847

The property is often referenced in relation to Exchange and IIS, which can require the size to be increased above the default value of 1024 bytes. The value must be some multiple of 1024; the previous example uses 8 KB, a commonly used size in response to a message where the request has exceeded the configured quota.

You may need to restart the WinRM service after changing the values:

```
Restart-Service winrm
```

Remoting and SSL

By default, Windows remoting requests are unencrypted. An HTTPS listener can be created to support encryption. Before attempting to create an HTTPS listener, a certificate is required.

Using a self-signed certificate is often the first step when configuring SSL. Windows 10 comes with a PKI module that can be used to create a certificate. The PKI module is only available in Windows PowerShell. In the following example, a self-signed certificate is created in the computer's personal store:

```
PS> New-SelfSignedCertificate -DnsName $env:COMPUTERNAME
PSParentPath: Microsoft.PowerShell.Security\Certificate::LocalMachine\MY
Thumbprint                               Subject
----------                               -------
D8D2F174EE1C37F7C2021C9B7EB6FEE3CB1B9A41 CN=SSLTEST
```

Once the certificate has been created, you can create an HTTPS listener using the `WSMan` drive. Replace the thumbprint in the following code with the thumbprint of a valid certificate, such as one created using the command in the previous example:

```
$params = @{
    Path                  = 'WSMan:\localhost\Listener'
    Address               = '*'
    Transport             = 'HTTPS'
    CertificateThumbprint = 'D8D2F174EE1C37F7C2021C9B7EB6FEE3CB1B9A41'
    Force                 = $true
}
New-Item @params
```

The `Force` parameter is used to suppress a confirmation prompt.

If Windows Firewall is running, a new rule must also be created to allow inbound connections to TCP port `5986`:

```
$params = @{
    DisplayName = $name = 'Windows Remote Management (HTTPS-In)'
    Name        = $name
    Profile     = 'Any'
    LocalPort   = 5986
    Protocol    = 'TCP'
}
New-NetFirewallRule @params
```

Set-WSManQuickConfig

Certificates used by remoting have the following requirements:

- The subject must contain the computer name (without a domain)
- The certificate must support the server authentication enhanced key usage
- The certificate must not be expired, revoked, or self-signed

If a certificate that meets these requirements is present, the `Set-WSManQuickConfig` command may be used:

```
Set-WSManQuickConfig -UseSSL
```

HTTPS listeners may be viewed as follows:

```
PS> Get-ChildItem WSMan:\localhost\Listener\* |
>>     Where-Object {
>>         (Get-Item "$($_.PSPath)\Transport").Value -eq 'HTTPS'
>>     }
WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Listener
Type      Keys                         Name
----      ----                         ----
Container {Transport=HTTPS, Address=*} Listener_1305953032
```

You can extend the preceding example by exploring the properties for the listener:

```
Get-ChildItem WSMan:\localhost\Listener | ForEach-Object {
    $listener = $_ | Select-Object Name
    Get-ChildItem $_.PSPath | ForEach-Object {
        $listener | Add-Member $_.Name $_.Value
    }
    $listener
} | Where-Object Transport -eq 'HTTPS'
```

The self-signed certificate can be assigned in this manner, but for an SSL connection to succeed, the client must trust the certificate. Without trust, the following error is shown:

```
PS> Invoke-Command -ScriptBlock { Get-Process } -ComputerName $env:COMPUTERNAME -UseSSL
[SSLTEST] Connecting to remote server SSLTEST failed with the following error message : The server
certificate on the destination computer (SSLTEST:5986) has the following errors:
The SSL certificate is signed by an unknown certificate authority. For more information, see the
about_Remote_Troubleshooting Help topic.
+ CategoryInfo : OpenError: (SSLTEST:String) [], PSRemotingTransportException
+ FullyQualifiedErrorId : 12175,PSSessionStateBroken
```

Several options are available to bypass this option:

- Disable certificate verification
- Add the certificate from the remote server to the local root certificate store

Disabling certificate verification can be achieved by configuring the options of a PSSession:

```
$options = New-PSSessionOption -SkipCACheck
$session = New-PSSession computerName -SessionOption $options
```

Either of the preceding options will allow the connection to complete. This can be verified using `Test-WSMan`:

```
Test-WSMan -UseSSL
```

If a new certificate is obtained, you can replace the certificate for the listener using `Set-Item`: the listener ID and certificate thumbprint should be updated with locally relevant values:

```
$params = @{
    Path  = 'WSMan:\localhost\Listener\Listener_1305953032\CertificateThumbprint'
    Value = 'D8D2F174EE1C37F7C2021C9B7EB6FEE3CB1B9A41'
}
Set-Item @params
```

Remoting and permissions

By default, PowerShell remoting (on Windows) requires administrative access or membership of the Remote Management Users. A summary of granted permissions may be viewed using `Get-PSSessionConfiguration`. The summary does not

include the permission level.

In PowerShell 7, the session configurations may be viewed using the following command:

```
Get-PSSessionConfiguration -Name PowerShell.7*
```

In Windows PowerShell, the name of the default configuration is `Microsoft.PowerShell`:

```
Get-PSSessionConfiguration -Name Microsoft.PowerShell
```

Windows PowerShell can view the session configurations for PowerShell 7, but PowerShell 7 is not able to view the session configuration for Windows PowerShell (at the time of writing).

Remoting permissions GUI

You can use the graphical interface to change permissions. The interface will be displayed when the following command is run:

```
Set-PSSessionConfiguration -Name PowerShell.7 -ShowSecurityDescriptorUI
```

Figure 14.1 displays a standard GUI for assigning permissions:

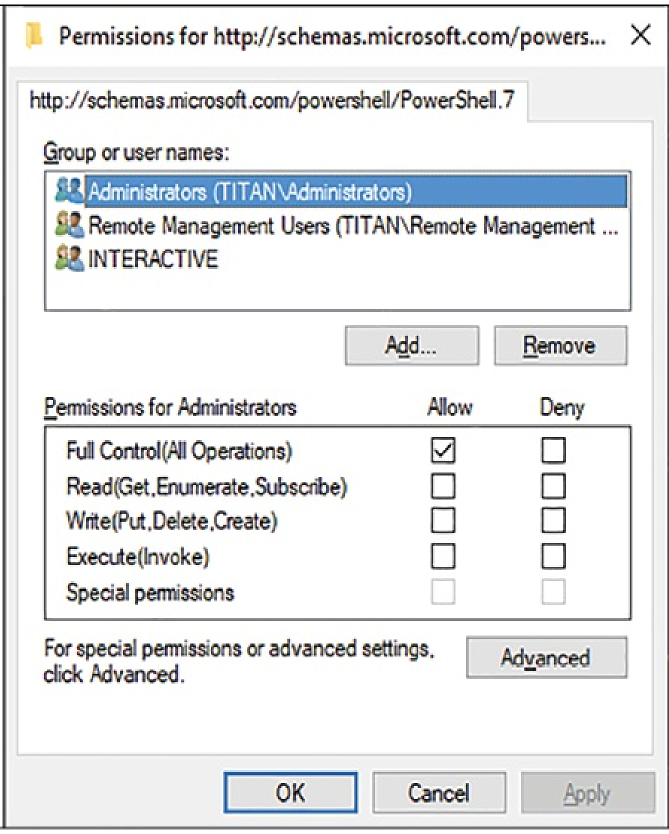


Figure 14.1: Permission settings

The session configuration defines four different permission levels:

- `Full`
- `Read`
- `Write`
- `Execute`

Remoting permissions by script

Permissions may also be changed using a script. The following commands retrieve the current security descriptor:

```
using namespace System.Security.AccessControl

$sddl = Get-PSSessionConfiguration PowerShell.7 |
    Select-Object -ExpandProperty SecurityDescriptorSddl
$acl = [CommonSecurityDescriptor]::new(
    $false,
    $false,
    $sddl
)
$acl.DiscretionaryAcl
```

Note that the name of the session configuration in the preceding example will vary depending on the version of PowerShell in use.

The object created here does not translate access masks into meaningful names. There are a small number of possible values for the access mask (shown here as 32-bit integers):

- `Full` (`All` operations): `268435456`
- `Read` (`Get` , `Enumerate` , `Subscribe`): `-2147483648`
- `Write` (`Put` , `Delete` , `Create`): `1073741824`
- `Execute` (`Invoke`): `536870912`

Permissions may be combined by using the `-bor` operator. For example, read and write may be defined using the following:

```
$readAndWrite = -2147483648 -bor 1073741824
```

Granting `Read` , `Write` , and `Execute` individually should be equivalent to `Full Control` . However, the result of binary (or the composite of all values) is `-536870912` , not the expected value for `Full` .

Understanding these values allows the current settings to be displayed in more detail than `Get-PSSessionConfiguration` displays. The function adds two script properties to each of the access control entries in the discretionary ACL. The first translates the `SID` into an account name; the second translates the access mask into a name (or set of names).

The example uses an enumeration (`enum`) to describe the possible access rights:

```
using namespace System.Security.AccessControl; using namespace System.Security.Principal

[Flags()]
enum SessionAccessRight {
    All     = -536870912
    Full    = 268435456
    Read    = -2147483648
    Write   = 1073741824
    Execute = 536870912
}

function Get-PSSessionAcl {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory)]
        [String]$Name
    )

    Get-PSSessionConfiguration -Name $Name | ForEach-Object {
        [CommonSecurityDescriptor]::new(
            $false,
            $false,
            $_.SecurityDescriptorSddl
        )
    }
}

function Get-PSSessionAccess {
    [CmdletBinding()]
    param (
        [String[]]$Name
    )
    Get-PSSessionConfiguration @PSBoundParameters | ForEach-Object {
        $sessionConfiguration = $_
        (Get-PSSessionAcl -Name $_.Name).DiscretionaryAcl |
            Select-Object -Property @(
                @{ Name = 'SessionName'; Expression = {
                    $sessionConfiguration.Name
                }}
                @{ Name = 'Identity';    Expression = {
                    $_.SecurityIdentifier.Translate([NTAccount])
                }}
                @{ Name = 'AccessRight'; Expression = {
                    [SessionAccessRight]$_.AccessMask
                }}
                '*'
            )
    }
}
```

Additional access may be granted by using the `AddAccess` method on `DiscretionaryAcl`. Granting access requires the `SID` of an account. The `SID` can be retrieved using the same `Translate` method that was used to get an account name from a `SID`. For example, the security identifier of the local administrator account may be retrieved as follows:

```
using namespace System.Security.Principal

([NTAccount]"Administrator").Translate([SecurityIdentifier])
```

Adding to the discretionary ACL may be achieved as shown in the following snippet. The example makes use of the `Get-PSSessionAcl` function and the `SessionAccessRight` enumeration created previously to grant access to the current user. The current user is identified using environment variables. The following example modifies the ACL for the `PowerShell.7` session; you can use the `Get-PSSessionConfiguration` command to show sessions if this is not present:

```
using namespace System.Security.AccessControl
using namespace System.Security.Principal
$sessionName = "PowerShell.7"
$identity = "$env:USERDOMAIN\$env:USERNAME"
$acl = Get-PSSessionAcl -Name $sessionName
$acl.DiscretionaryAcl.AddAccess(
    'Allow',
    ([NTAccount]$identity).Translate([SecurityIdentifier]),
    [Int][SessionAccessRight]'Full',
    'None',   # Inheritance flags
    'None'    # Propagation flags
)
```

The updated ACL must be converted back to an SDDL string to apply the change:

```
$sddl = $acl.GetSddlForm('All')
Set-PSSessionConfiguration Microsoft.PowerShell -SecurityDescriptorSddl $sddl
```

User Account Control

User Account Control (**UAC**) restricts local (not domain) user accounts that log on using a remote connection. By default, the remote connection will be made as a standard user account, that is, a user without administrative privileges.

The `Enable-PSRemoting` command disables UAC remote restrictions. If another method has been used to enable remoting, and a local account is being used to connect, it is possible that remote restrictions are still in place.

The current value can be viewed using the following:

```
$params = @{
    Path = 'HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System'
    Name = 'LocalAccountTokenFilterPolicy'
}
Get-ItemPropertyValue @params
```

If the key or value is missing, an error will be thrown. The impact of this is described in the help article linked below. UAC remote restrictions can be disabled as follows. Using the `Force` parameter will allow the creation of both the key and the value:

```
$params = @{
    Path  = 'HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System'
    Name  = 'LocalAccountTokenFilterPolicy'
    Value = 1
    Force = $true
}
Set-ItemProperty @params
```

The change used previously, and UAC remote restrictions, are described in the following Microsoft Knowledge Base article 951016:

https://support.microsoft.com/help/951016/description-of-user-account-control-and-remote-restrictions-in-windows

Trusted hosts

If a remote system is not part of a domain, or is part of an untrusted domain, an attempt to connect using remoting may fail. The remote system must either be listed in trusted hosts or use SSL.

The use of trusted hosts also applies when connecting to another computer using a local user account.

Trusted hosts are set on the client, that is, the system making the connection. The following command gets the current value:

```
Get-Item WSMan:\localhost\Client\TrustedHosts
```

The value is a comma-delimited list. Wildcards are supported in the list. The following function may be used to add a value to the list:

```
function Add-TrustedHost {
    param (
        [String]$Hostname
    )

    $item = Get-Item WSMan:\localhost\Client\TrustedHosts
    $trustedHosts = @($item.Value -split ',')
    $trustedHosts = $trustedHosts + $Hostname |
        Where-Object { $_ } |
        Select-Object -Unique

    $item | Set-Item -Value ($trustedHosts -join ',')
}
```

In the next section, you will become familiar with PSSessions, which can be created from one host to another to allow commands to be executed.

PSSessions

PSSessions use PowerShell remoting to communicate between servers. PSSessions can be used for anything from remote commands and script execution to providing a remote shell.

By default, PSSessions use the `Microsoft.PowerShell` configuration, described by the built-in `$PSSessionConfigurationName` variable. Remoting connections therefore always default to connecting to Windows PowerShell. Administrative rights are required to view and change session configuration information.

If you are creating a session to the local system, the `-EnableNetworkAccess` parameter should be added to the following commands. This parameter is only applicable to sessions that are created from and connect to the same system.

New-PSSession

Sessions are created using the `New-PSSession` command. In the following example, a session is created on a computer named `PSTEST`:

```
PS> New-PSSession -ComputerName PSTEST
Id   Name       ComputerName  State    ConfigurationName      Availability
--   ----       ------------  -----    -----------------      ------------
1    Session1   PSTEST        Opened   Microsoft.PowerShell   Available
```

Get-PSSession

Sessions created using `New-PSSession` persist until the PSSession is removed (by `Remove-PSSession`) or the PowerShell session ends. The following example returns sessions created in the current PowerShell session:

```
PS> Get-PSSession | Select-Object Id, ComputerName, State
Id ComputerName State
-- ------------ -----
 1 PSTEST       Opened
```

If the `ComputerName` parameter is supplied, `Get-PSSession` will show sessions created on that computer. For example, imagine a session is created in one PowerShell console:

```
$session = New-PSSession -ComputerName PSTest -Name Example
```

A second administrator console session will be able to view details of that session:

```
PS> Get-PSSession -ComputerName PSTest |
>>     Select-Object Name, ComputerName, State
Name     ComputerName  State
----     ------------  -----
Example  PSTest        Disconnected
```

Invoke-Command

`Invoke-Command` may be used with a PSSession to execute a command or script on a remote system:

```
$session = New-PSSession -ComputerName $env:COMPUTERNAME
Invoke-Command { Get-Process -Id $PID } -Session $session
```

$env:COMPUTERNAME is localhost

Connecting to a session requires administrative access by default. The preceding command will fail if PowerShell is not running with an administrative token (run as administrator).

A PowerShell session with the administrator token can be started using the `Start-Process powershell -Verb RunAs` command.

`Invoke-Command` has a wide variety of different uses, as shown in the command help. For example, a single command can be executed against a list of computers; the command is run against each computer in turn:

```
Invoke-Command { Get-Process -Id $PID } -ComputerName 'first', 'second', 'third'
```

This technique can be useful when combined with `AsJob`. Pushing the requests into the background allows each server to get on with its work, pushing it back when the work is complete.

Once the job created by the previous command has completed, any data may be retrieved using the `Receive-Job` command.

Several advanced techniques may be used with `Invoke-Command`, which are described in the following sections.

Local functions and remote sessions

The following example executes a function created on the local machine in a remote system using positional arguments:

```
function Get-FreeSpace {
    param (
        [Parameter(Mandatory = $true)]
        [String]$Name
    )

    [Math]::Round((Get-PSDrive $Name).Free / 1GB, 2)
}
Invoke-Command ${function:Get-FreeSpace} -Session $session -ArgumentList C
```

This technique succeeds because the body of the function is declared as a script block. `ArgumentList` is used to pass a positional argument into the `DriveLetter` parameter.

If the function depends on other locally defined functions, the attempt will fail.

Using splatting with ArgumentList

The `ArgumentList` parameter of `Invoke-Command` does not offer a means of passing named arguments to a command.

The following example uses splatting to pass parameters. The function is defined on the local system, and the definition of the function is passed to the remote system:

```
# A function which exists on the current system
function Get-FreeSpace {
    param (
        [Parameter(Mandatory = $true)]
        [String]$Name
    )

    [Math]::Round((Get-PSDrive $Name).Free / 1GB, 2)
}

# Define parameters to pass to the function
$params = @{
    Name = 'c'
}

# Execute the function with a named set of parameters
Invoke-Command -ScriptBlock {
    param ( $definition, $params )

    & ([ScriptBlock]::Create($definition)) @params
} -ArgumentList ${function:Get-FreeSpace}, $params -ComputerName $computerName
```

In the preceding example, the definition of the `Get-FreeSpace` function is passed as an argument along with the requested parameters. The script block used with `Invoke-Command` converts the definition into a `ScriptBlock` and executes it.

The AsJob parameter

The `AsJob` parameter can be used with `Invoke-Command`, for example:

```
$session = New-PSSession PSTest
Invoke-Command -Session $session -AsJob -ScriptBlock {
    Start-Sleep -Seconds 120 'Done sleeping'
}
```

The command finishes immediately and returns the job that has been created.

While the job is running, the session availability is set to `Busy`:

```
PS> $session | Select-Object Name, ComputerName, Availability
Name      ComputerName  Availability
----      ------------  ------------
Session1  PSTest        Busy
```

Attempts to run another command against the same session will result in an error message.

Once the job has completed, the `Receive-Job` command may be used.

Disconnected sessions

The `InDisconnectedSession` of `Invoke-Command` starts the requested script and immediately disconnects the session. This allows a script to be started and collected from a different console session or a different computer.

The `session` parameter cannot be used with `InDisconnectedSession`; `Invoke-Command` creates a new session for a specified computer name. The session is returned by the following command:

```
Invoke-Command {
    Start-Sleep -Seconds 120
    'Done'
} -ComputerName PSTest -InDisconnectedSession
```

A second PowerShell session or computer can connect to the disconnected session to retrieve the results. The following command assumes that only one session exists with the `PSTest` computer:

```
Get-PSSession -ComputerName PSTest |
    Connect-PSSession |
    Receive-PSSession
```

Tasks started with `AsJob` will also continue to run if a session is disconnected. The following example creates a session, starts a long-running process, and disconnects the session:

```
$session = New-PSSession PSTest -Name 'Example'
Invoke-Command { Start-Sleep -Seconds (60 * 60) } -Session $session -AsJob
Disconnect-PSSession $session
```

Once the session has been created and disconnected, the PowerShell console can be closed. A second PowerShell console can find and connect to the existing session:

```
$session = Get-PSSession -ComputerName PSTest -Name 'Example'
Connect-PSSession $session
```

Reviewing the details of the session will show that it is busy running `Start-Sleep`:

```
PS> Get-PSSession | Select-Object Name, ComputerName, State, Availability
Name      ComputerName   State    Availability
----      ------------   -----    ------------
Example   PSTest         Opened   Busy
```

The using variable scope

When working with `Invoke-Command`, PowerShell makes the `using` variable scope modifier available.

The `using` variable scope allows access to variables created on a local machine within a script block used with `Invoke-Command`.

The following example shows the use of a variable that contains parameters for `Get-Process`. The local variable may contain any reasonable value:

```
$params = @{
    Name            = 'powershell'
    IncludeUserName = $true
}
Invoke-Command -ComputerName PSTest -ScriptBlock {
    $params = $using:params
    Get-Process @params
}
```

The `using` scope is a handy alternative to the `ArgumentList` parameter.

The Enter-PSSession command

`Enter-PSSession` may be employed to use a session as a remote console. By default, `Enter-PSSession` accepts a computer name as the first argument:

```
Enter-PSSession PSTest
```

In a similar way, an existing session might be used:

```
$session = New-PSSession -ComputerName PSTest
Enter-PSSession -Session $session
```

`Enter-PSSession` uses WS-Management as a means of exchanging information between the client and the server. Once a command is typed and the return key is pressed, the entire command is sent to the remote host. The result of the command is sent back using the same mechanism. This exchange can inject a small amount of latency into the shell.

`Enter-PSSession` is for interactive use only; it cannot be included in scripts except as the very last command.

Import-PSSession

`Import-PSSession` brings commands from a remote computer into the current session. Microsoft Exchange uses this technique to provide remote access to the Exchange Management Shell, and PowerShell 7 uses this technique to provide access to modules compatible with Windows PowerShell only.

The following example imports the `NetAdapter` module from a remote server into the current session:

```
$computerName = 'PSTest'
$session = New-PSSession -ComputerName $computerName
Import-PSSession -Session $session -Module NetAdapter
```

Any commands used within this module are executed against the session target, not against the local computer.

If the session is removed, the imported module and its commands will be removed from the local session.

Export-PSSession

In the preceding example, `Import-PSSession` is used to immediately import commands from a remote system into a local session. `Export-PSSession` writes a persistent module that can be used to achieve the same goal.

The following example creates a module in the current user's module path:

```
$computerName = 'PSTest'
$session = New-PSSession -ComputerName $computerName
Export-PSSession -Session $session -Module NetAdapter -OutputModule "NetAdapter-$computerName"
```

Once the module has been created, it can be imported by name:

```
Import-Module "NetAdapter-$computerName"
```

This process replaces the need to define and import a session and is useful for remote commands that are used on a regular basis.

Copying items between sessions

PowerShell 5 introduced the ability to copy between sessions using the `Copy-Item` command.

The `FromSession` parameter is used to copy a file to the local system:

```
$session1 = New-PSSession PSTest1
Copy-Item -Path C:\temp\doc.txt -Destination C:\Temp -FromSession $session1
```

In the preceding example, `Path` is on `PSTest1`.

The `ToSession` parameter is used to copy a file to a remote system:

```
$session2 = New-PSSession PSTest2
Copy-Item -Path C:\temp\doc.txt -Destination C:\Temp -ToSession $session2
```

In the previous example, the path used for the destination parameter is on `PSTest2`.

The `FromSession` and `ToSession` parameters cannot be specified together; two separate commands are required to copy a file between two remote sessions.

PowerShell 6 introduced the ability to use PowerShell remoting with Linux systems. Remoting must be configured on Linux before it can be used.

Remoting on Linux

Microsoft provides instructions for installing PowerShell on Linux; these should be followed before attempting to configure remoting:

https://docs.microsoft.com/powershell/scripting/install/installing-powershell-core-on-linux

Once installed, it is possible to make PowerShell the default shell. This is optional and does not affect remoting. First, check that PowerShell is listed in the `shells` file:

```
Get-Content /etc/shells   # Use cat or less in Bash
```

The native `chsh` (change shell) command can be used to change the default shell for the current user, as shown in the following example:

```
chsh -s /usr/bin/pwsh
```

To configure remoting using WSMan, the OMI and PSRP packages must be installed. The following example uses `yum` since the operating system in use is CentOS 7:

```
yum install omi.x86_64 omi-psrp-server.x86_64
```

By default, CentOS has a firewall configured. The network interface in use, in this case, `eth0`, must be added to an appropriate zone, and WinRM must be allowed:

```
firewall-cmd --zone=home --change-interface=eth0
firewall-cmd --zone=home --add-port=5986/tcp
```

Once configured, it should be possible to connect to the remote host. SSL is required to form the connection. The certificate is self-signed so certificate validity tests must be skipped at this stage:

```
$params = @{
    ComputerName   = 'LinuxSystemNameOrIPAddress'
    Credential     = Get-Credential
    Authentication = 'Basic'
    UseSsl         = $true
    SessionOption  = New-PSSessionOption -SkipCACheck -SkipCNCheck
}
Enter-PSSession @params
```

The state of the certificate leaves the identity of the host in question, but it does ensure that traffic is encrypted. If SSL is to be used beyond testing, a valid certificate chain should be established.

At this point, the remote computer should be accessible using both Windows PowerShell and PowerShell Core.

Remoting over SSH

PowerShell Core introduced the concept of remoting over SSH. This provides a useful alternative to remoting over HTTPS, avoiding the burden of managing certificates:

https://github.com/PowerShell/PowerShell/tree/866b558771a20cca3daa47f300e830b31a24ee95/docs/new-features/remoting-over-ssh

The SSH transport for remoting cannot be used from Windows PowerShell, only PowerShell 6 and 7.

Connecting from Windows to Linux

If connecting from Windows, an SSH client must be installed.

In Windows 10 and Windows Server 2019, you can enable OpenSSH as an optional feature. For example:

```
Add-WindowsCapability -Online -Name OpenSSH.Client~~~~0.0.1.0
```

The feature is documented on Microsoft Docs:

https://docs.microsoft.com/windows-server/administration/openssh/openssh_install_firstuse

Alternatively, the `openssh` package can be installed using the Chocolatey package manager (http://chocolatey.org):

```
choco install openssh
```

Depending on the desired configuration, public key authentication may be enabled in the SSH daemon configuration file. A subsystem must be added to the file.

To enable public key authentication, set `PubkeyAuthentication`:

```
PubkeyAuthentication yes
```

An existing subsystem entry will likely exist toward the end of the file; this new entry can be added beneath the existing entry (the value in the following code is a single configuration line):

```
Subsystem       powershell     /opt/microsoft/powershell/7/pwsh -sshs -NoLogo -NoProfile
```

The `sshd` service should be restarted after changing the configuration file:

```
service sshd restart
```

The connection in this example uses SSH key authentication. This requires an SSH key on Windows. If an existing key is not available, the `ssh-keygen` command can be used to create a new key pair. The command will prompt for any information it requires.

The private key created by this command will be used when connecting to a remote host. The public key is used to authorize a user and will be placed on the Linux system.

The public key can be obtained by running the following command on the system on which it was generated. This command assumes default filenames were used when generating the key:

```
Get-Content ~\.ssh\id_rsa.pub | Set-Clipboard
```

~ is home

The tilde character may be used as shorthand for the path to the home directory. On Linux it is typically

> `/home/<username>`, and on Windows it is typically similar to `C:\users\<username>`.
>
> `~` may be replaced with the `$home` variable, or the `$env:USERPROFILE` environment variable on Windows, if desired.

The public key should be added to the `authorized_keys` files on Linux:

```
$publicKey = 'ssh-rsa AAAABG...'
New-Item ~/.ssh -ItemType Directory
Set-Content -Path ~/.ssh/authorized_keys -Value $publicKey
```

Once complete, a session can be created and used to interact with the Linux system:

```
$params = @{
    HostName     = 'LinuxSystemNameOrIPAddress'
    UserName     = $env:USERNAME
    SSHTransport = $true
    KeyFilePath  = '~\.ssh\id_rsa'
}
Enter-PSSession @params
```

As remoting is encapsulated in exchanged XML requests, interactive commands such as `vi` will not work in the remoting session.

Connecting from Linux to Windows

Connecting from Linux to Windows is a harder path; it is clearly undergoing rapid change and is much less mature than connections in the other direction.

Before moving on to configuring SSH, verify that `WSMan` functions. An HTTPS listener must be set up; HTTP connections are prohibited by the PSRP package. If HTTPS is not already available, a self-signed certificate may be created and used as shown in the *Remoting and SSL* section.

If remoting is not yet configured for PowerShell 7, run the `Enable-PSRemoting` command in the Core console (as an administrator). Once enabled, find the name of the configuration entry using the `Get-PSSessionConfiguration` command.

You can use the configuration name to create a session to PowerShell Core that runs on the Windows system:

```
$params = @{
    HostName          = 'WindowsSystemNameOrIPAddress'
    Credential        = (Get-Credential)
    Authentication    = 'Basic'
    UseSSL            = $true
    ConfigurationName = 'PowerShell.6.1.1'
}
Enter-PSSession @params
```

At the time of writing, attempting to connect from Linux to a PowerShell 5.1 session results in an "access denied" error message.

The OpenSSH package must be installed on Windows to continue. The OpenSSH server feature can be installed using the following command:

```
Add-WindowsCapability -Online -Name OpenSSH.Server~~~~0.0.1.0
```

Alternatively, if the Chocolatey package was used, the server can be enabled:

```
& "C:\Program Files\OpenSSH-Win64\install-sshd.ps1"
```

Windows Firewall must also be opened if it is in use:

```
$params = @{
    DisplayName = 'OpenSSH Server (sshd)'
    Name        = 'sshd'
    Enabled     = $true
    Direction   = 'Inbound'
    LocalPort   = 22
    Protocol    = 'TCP'
    Action      = 'Allow'
}
New-NetFirewallRule @params
```

Once this step is complete, it should be possible to create an SSH connection from Linux to Windows:

```
ssh user@WindowsSystemNameOrIPAddress
```

As with configuring Linux, public key authentication may be allowed, and a subsystem must be configured, this time on the Windows system. The `C:\ProgramData\ssh\sshd_config` file must be edited.

To enable public key authentication, set `PubkeyAuthentication`:

```
PubkeyAuthentication yes
```

Add a subsystem to the file. This may be specified in addition to any existing subsystem:

```
Subsystem    powershell    C:/progra~1/PowerShell/7/pwsh.exe -sshs -NoLogo -NoProfile
```

PowerShell 7 may be configured as the default shell for incoming SSH connections:

```
$params = @{
    Path = 'HKLM:\SOFTWARE\OpenSSH'
    Name = 'DefaultShell'
    Value = Join-Path -Path $pshome -ChildPath 'pwsh.exe'
    Force = $true
}
New-ItemProperty @params
```

The `sshd` service should be restarted after changing the configuration file or the default shell:

```
Restart-Service sshd
```

At this point, it will be possible to create a remoting session using SSH, by entering a password when prompted:

```
$params = @{
    HostName     = 'WindowsSystemNameOrIPAddress'
    UserName     = $env:USERNAME
    SSHTransport = $true
}
Enter-PSSession @params
```

Public key authentication may be configured in the same way as was done for Linux. A key can be generated on Linux using the `ssh-keygen` command.

The public key, by default `~/.ssh/id_rsa.pub`, may be added to an `authorized_keys` file on Windows. The following command, when run on Linux, displays the public key:

```
Get-Content ~/.ssh/id_rsa.pub
```

This public key may be added to an `authorized_keys` file for a user on the Windows system:

```
$publicKey = 'ssh-rsa AAAABG...'
Set-Content -Path ~/.ssh/authorized_keys -Value $publicKey
```

At this point, the Linux system will be able to use public key authentication to access the Windows system:

```
$params = @{
    HostName     = 'WindowsSystemNameOrIPAddress'
    UserName     = 'username'
    SSHTransport = $true
    KeyFilePath  = '~\.ssh\id_rsa'
}
Enter-PSSession @params
```

Extending this further, Windows systems running PowerShell 7 and the SSH daemon may use SSH as a remoting transport to access other Windows systems.

One of the most common problems when using remoting is accessing remote resources from within a remoting session.

The double-hop problem

The double-hop problem describes a scenario in PowerShell where remoting is used to connect to a host and the remote host tries to connect to another resource. In this scenario, the second connection, the second hop, fails because authentication cannot be implicitly passed.

Over the years, there have been numerous articles that discuss this problem. Ashley McGlone published a blog post in 2016 that describes the problem and the possible solutions:

https://docs.microsoft.com/archive/blogs/ashleymcglone/powershell-remoting-kerberos-double-hop-solved-securely

This section briefly explores using `CredSSP`, as well as how to pass explicit credentials to a remote system. Neither of these options is considered secure, but they require the least amount of work to implement.

These two options are useful in the following situations:

- The remote endpoint is trusted and has not been compromised
- Critical authentication tokens can be extracted by an administrator on the remote system
- They are not used for wide-scale regular or scheduled automation, as the methods significantly increase exposure

CredSSP

Credentials passed using `CredSSP` are sent in clear text. `CredSSP` is not considered secure.

A session can be created using `CredSSP` as the authentication provider:

```
$params = @{
    ComputerName   = 'PSTest'
    Credential     = Get-Credential
    Authentication = 'CredSSP'
}
New-PSSession @params
```

`CredSSP` must be enabled on the client to support passing credentials to a remote system. The `DelegateComputer` parameter

can be used with either a specific name or a wildcard (`*`):

```
Enable-WSManCredSSP -Role Client -DelegateComputer PSTest
```

`CredSSP` must also be enabled on the server to receive credentials:

```
Enable-WSManCredSSP -Role Server
```

If this approach is used as a temporary measure, the `CredSSP` roles might be removed afterward.

On the server making the connection, the `Client` role can be disabled:

```
Disable-WSManCredSSP -Role Client
```

On the remote system, the `Server` role can be disabled:

```
Disable-WSManCredSSP -Role Server
```

Passing credentials

Passing credentials into a remote session means the second hop can authenticate without being dependent on authentication tokens from the original system.

In this example, the `using` variable scope is used to access a credential variable. The credential is used to run a query against Active Directory from a remote system:

```
$Credential = Get-Credential
Invoke-Command -ComputerName PSTest -ScriptBlock {
    Get-ADUser -Filter * -Credential $using:Credential
}
```

Passing credentials in this manner works around the problem but cannot be considered a secure solution. The article quoted at the beginning of this session properly explores the alternatives available.

PSSessions are not the only thing to use Windows remoting. **Common Information Model** (**CIM**) sessions also use Windows remoting but are only used to access WMI.

CIM sessions

CIM sessions are used to work with CIM services, predominantly WMI or commands that base their functionality on WMI. Such commands include those in the `NetAdapter` and `Storage` modules available on Windows 2012 and Windows 8. A list of commands that support CIM sessions may be viewed by entering the following:

```
Get-Command -ParameterName CimSession
```

The list will only include commands from modules that have been imported.

The CIM commands are only available in Windows installations of PowerShell. The CIM implementation is specific to the Microsoft platform, and is not implemented only in .NET.

New-CimSession

CIM sessions are created using the `New-CimSession` command. The following example creates a CIM session using the current system as the computer name and using `WSMan` as the protocol:

```
PS> New-CimSession -ComputerName $env:COMPUTERNAME
Id           : 1
Name         : CimSession1
InstanceId   : bc03b547-1051-4af1-a41d-4d16b0ec0402
ComputerName : PSTEST
Protocol     : WSMAN
```

If the computer name parameter is omitted, the protocol will be set to `DCOM`:

```
PS> New-CimSession
Id           : 2
Name         : CimSession2
InstanceId   : 804595f4-0144-4590-990a-92b2f22f894f
ComputerName : localhost
Protocol     : DCOM
```

`New-CimSession` can be used to configure operation timeout settings and whether an initial network test should be performed.

The protocol used by `New-CimSession` can be changed using `New-CimSessionOption`. Changing the protocol can be useful if there is a need to interact with systems where WinRM is not running or configured:

```
PS> $params = @{
>>     ComputerName = $env:COMPUTERNAME
>>     SessionOption = New-CimSessionOption -Protocol DCOM
>> }
PS> New-CimSession @params
Id           : 3
Name         : CimSession3
InstanceId   : 29bba117-c899-4389-b874-5afe43962a1e
ComputerName : PSTEST
Protocol     : DCOM
```

Get-CimSession

Sessions created using `New-CimSession` persist until the CIM session is removed (by `Remove-CimSession`) or the PowerShell session ends:

```
PS> Get-CimSession | Select-Object Id, ComputerName, Protocol
Id   ComputerName Protocol
--   ------------ --------
 1   PSTEST       WSMAN
 2   localhost    DCOM
 3   PSTEST       DCOM
```

Using CIM sessions

Once a CIM session has been created, it can be used for one or more CIM requests. In the following example, a CIM session is created and then used to gather disk and partition information:

```
$ErrorActionPreference = 'Stop'
try {
    $session = New-CimSession -ComputerName $env:COMPUTERNAME
    Get-Disk -CimSession $session
    Get-Partition -CimSession $session
} catch {
    throw
}
```

In the preceding script, if the attempt to create the session succeeds, the session will be used to get disk and partition information.

Error handling with `try` and `catch` is discussed in *Chapter 22, Error Handling*. The block is treated as a transaction; if a single command fails, the block will stop running. If the attempt to create a new session fails, `Get-Disk` and `Get-Partition` will not run.

Just Enough Administration

Just Enough Administration (**JEA**) leverages PowerShell remoting to allow administrative delegation via a remoting session.

JEA consists of:

- A session configuration file that describes the commands to be made available and language modes
- A registered PSSession configuration that is created based on that file
- Access control that is set on the PSSession configuration

JEA documentation can be found on Microsoft Docs:

https://docs.microsoft.com/powershell/scripting/learn/remoting/jea/overview?view=powershell-7

JEA configuration is defined in a session configuration file. The file is saved as a PowerShell Data File (a `PSSC` file, the same format as `psd1` files) and is used to define and register the JEA remoting endpoint.

Microsoft has a couple of small examples that can be viewed on GitHub:

https://github.com/PowerShell/JEA

JEA can be configured manually, as shown in the following sections, or using a DSC configuration in Windows PowerShell.

Session configuration

Session configuration files can be created using the `New-PSSessionConfigurationFile` command.

The session configuration file determines which commands are available to anyone importing the session.

A session configuration file should be restrictive. Commands such as `Invoke-Expression` and `Add-Type` should not be permitted; they allow arbitrary code execution, which defeats the point of a restricted endpoint. The language mode will ideally be set to restricted or none.

The following example is a simple session configuration that allows access to the `Get-ComputerInfo` command. Each of the commands is executed in an administrative session in PowerShell 7:

```
if (-not (Test-Path c:\jea)) {
    New-Item c:\jea -ItemType Directory
}
$params = @{
    Path           = 'c:\jea\jea.pssc'
    LanguageMode   = 'NoLanguage'
    SessionType    = 'RestrictedRemoteServer'
    VisibleCmdlets = @(
        'Get-ComputerInfo'
        'Export-Csv'
    )
}
New-PSSessionConfigurationFile @params
```

Once the configuration file has been created the session configuration can be registered:

```
$params = @{
    Name = 'JEATest'
    Path = 'c:\jea\jea.pssc'
}
Register-PSSessionConfiguration @params
```

The file that describes the session can be deleted once the session is registered.

You can use the new session immediately, for example, by entering the possession:

```
Enter-PSSession -ComputerName localhost -ConfigurationName JEATest
```

By default, the session configuration allows administrators or members of the local group Remote Management Users. Additional rights may be granted as seen when exploring remoting permissions.

Once inside the session, several default commands will be available, such as `Get-Command`. These are permitted because of the `SessionType`. `Export-Csv` and `Get-ComputerInfo` are available because they have been explicitly permitted.

If the endpoint is no longer required, it can be removed as shown in the following code:

```
Unregister-PSSessionConfiguration -Name 'JEATest'
```

Careful consideration should be given to which commands are exposed by JEA. The default account configuration used in the preceding example grants administrative rights to the system hosting the endpoint.

Role capabilities

Role capabilities add a lot of flexibility to the endpoint. Different commands can be made available via an endpoint depending on the user or a group the user belongs to. If a user belongs to more than one role the user will be granted access to the commands from each role.

Role capabilities must be maintained in a file. A role capabilities file can be created using the `New-PSRoleCapabilityFile` command:

```
$params = @{
    Path = 'c:\jea\group.psrc'
    VisibleCmdlets  = @(
        'Get-ComputerInfo'
        'Export-Csv'
    )
}
New-PSRoleCapabilityFile @params
```

This file may be used when creating a session configuration in place of the simple list of commands:

```
$params = @{
    Path            = 'c:\jea\jea.pssc'
    LanguageMode    = 'NoLanguage'
    SessionType     = 'RestrictedRemoteServer'
    RoleDefinitions = @{
        'DOMAIN\Group' = @{
            RoleCapabilityFiles = 'c:\jea\group.psrc'
        }
    }
}
New-PSSessionConfigurationFile @params
```

The role capability file must be retained after the session has been registered.

With careful consideration, JEA can be an effective delegation tool. Thought must be given to the permitted language features and the permissions assigned to accounts.

Summary

In this chapter, we delved into remoting in PowerShell, starting with WS-Management, and looked at the SSH transport features in PowerShell 7.

The double-hop problem was introduced, along with several possible ways to work around the issue. Finally, we explored CIM sessions and JEA.

The next chapter explores asynchronous processing.

15

Asynchronous Processing

PowerShell prefers to run things synchronously, that is, sequentially, or one after another. However, it is frequently necessary to run many things simultaneously, without waiting for another command to complete. This is known as an **asynchronous operation**.

Operations of this nature may be local to the current machine or might run queries or code against remote systems.

PowerShell includes several different commands and classes that can do more than one thing at a time. The most obvious of these are the job commands.

In addition to the job commands, PowerShell can react to .NET events and use runspaces and runspace pools.

This chapter explores the following topics:

- Working with jobs
- Reacting to events
- Using runspaces and runspace pools

PowerShell jobs, using commands like `Start-Job`, are the simplest starting point for executing code asynchronously.

Working with jobs

The `Start-Job` command in PowerShell provides a means of executing code asynchronously by creating a new PowerShell process for each job.

As each job executes within a new process, data cannot be shared between jobs. Any required modules, functions, or variables all need to be imported into each job.

Additionally, jobs might be considered resource heavy as each job must start both a PowerShell process and a console window's host process.

PowerShell provides several commands to create and interact with jobs. In addition to the following commands, you can use `Invoke-Command` with the `AsJob` parameter when acting against remote systems.

The Start-Job, Get-Job, and Remove-Job commands

You can use the `Start-Job` command to execute a script block in a similar manner to `Invoke-Command`, as shown in *Chapter 14, Remoting and Remote Management*. Also, you can use `Start-Job` to execute a script using the `FilePath` parameter.

When `Start-Job` is executed, a job object, `System.Management.Automation.PSRemotingJob` is created. The job object continues to be available using the `Get-Job` command regardless of whether the output from `Start-Job` is assigned:

```
PS> Start-Job -ScriptBlock { Start-Sleep -Seconds 10 }
Id  Name  PSJobTypeName  State    HasMoreData  Location   Command
--  ----  -------------  -----    -----------  --------   -------
 1  Job1  BackgroundJob  Running        True   localhost  Start-Sleep -Seconds 10
PS> Get-Job
Id  Name  PSJobTypeName  State    HasMoreData  Location   Command
--  ----  -------------  -----    -----------  --------   -------
 1  Job1  BackgroundJob  Running         True  localhost  Start-Sleep -Seconds 10
```

When you work with a script using jobs, the common practice is to capture the jobs created instead of relying entirely on `Get-`

`Job`. This avoids problems if the module used in a script also creates jobs. The state of the job is reflected on the job object; `Get-Job` is not required to update the status.

Job objects and any data the job has returned remain available until you remove them using the `Remove-Job` command.

`Start-Job` includes a `RunAs32` parameter to run code on the 32-bit version of PowerShell if required.

`Start-Job` does not offer a throttling capability, that is, limiting the number of concurrent activities. PowerShell will simultaneously execute every job. Each job will compete for system resources. A `while` or `do` loop may be implemented to maintain a pool of running jobs:

```
$listOfJobs = 1..50
foreach ($job in $listOfJobs) {
    while (@(Get-Job -State Running).Count -gt 10) {
        Start-Sleep -Seconds 10
    }
    Start-Job {
        Start-Sleep -Seconds (Get-Random -Minimum 10 -Maximum 121)
    }
}
```

The jobs created here do not return any data and can therefore be removed as soon as they have been completed. Data must be retrieved from a job before it is removed.

The Receive-Job command

`Receive-Job` is used to retrieve data from a job. `Receive-Job` may be used when a job is executing, or when the job is finished. If `Receive-Job` is run before a job is finished, any existing values will be returned. Running `Receive-Job` again will get any new values that have been added since it was last run. This is shown in the following example:

```
$job = Start-Job {
    1..10 | ForEach-Object {
        $_
        Start-Sleep -Seconds 2
    }
}
Write-Host 'Sleeping 2'
Start-Sleep -Seconds 2
$job | Receive-Job
Write-Host 'Sleeping 5'
Start-Sleep -Seconds 5
$job | Receive-Job
```

The results from the script are received as they become available:

```
Sleeping 2
1
2
Sleeping 5
3
4
```

The remaining results will be available to `Receive-Job` as they are returned, or when the job has completed.

The `-Wait` parameter of `Receive-Job` will receive data from the job as it becomes available and send it to the output pipeline.

The Wait-Job command

The `Wait-Job` command waits for all the jobs in the input pipeline to complete. `Wait-Job` supports a degree of filtering and

offers a timeout parameter to determine the length of time to wait for a job to complete before the job is run in the background.

In some cases, it is desirable to pull off output from jobs as they complete. This can be solved by creating a `while` or `do` loop in PowerShell, reacting to jobs as the state changes:

```
while (Get-Job -State Running) {
    $jobs = Get-Job -State Completed
    $jobs | Receive-Job
    $jobs | Remove-Job
    Start-Sleep -Seconds 1
}
```

A `while` loop does not have an output pipeline; if output is to be piped to another command, it would need to be piped within the loop. For example, if the job output were filling a CSV file, `Export-Csv` would be added inside the loop and the `Append` parameter would be used:

```
while (Get-Job -State Running) {
    $jobs = Get-Job -State Completed
    $jobs | Receive-Job | Export-Csv output.csv -Append
    $jobs | Remove-Job
    Start-Sleep -Seconds 1
}
```

This technique is useful if the job is returning a large amount of data. Streaming output to a file as jobs complete will potentially help manage memory usage across a larger number of jobs.

This approach can be combined with the snippet, which limits the number of concurrent jobs. The tweak is shown as follows:

```
$listOfJobs = 1..50
$jobs = foreach ($job in $listOfJobs) {
    while (@(Get-Job -State Running).Count -gt 10) {
        Start-Sleep -Seconds 10
    }
    Start-Job {
        Start-Sleep -Seconds (Get-Random -Minimum 10 -Maximum 121)
    }
    Get-Job -State Completed | Receive-Job | Export-Csv output.csv -Append
}
$jobs | Wait-Job | Receive-Job | Export-Csv output.csv -Append
```

The final line is required to wait for and then receive the jobs that were still running when the last job was started.

Jobs and the using scope modifier

The `$using` scope modifier was introduced in *Chapter 3, Working with Objects in PowerShell*, while exploring the new `Parallel` parameter for the `ForEach-Object` command.

All commands that use jobs can make use of the `$using` scope modifier to access variables created in the parent scope. `Invoke-Command` is also able to leverage `$using`, even if it is not using a job.

For example, the following job uses the value of a variable from the scope that started the job:

```
PS> $message = 'Hello world'
PS> Start-Job -ScriptBlock { Write-Host $using:message } |
>>     Receive-Job -Wait
Hello world
```

You can use the `$using` scope modifier to access variables in other scopes. For example, a variable explicitly created in the global scope:

```
function Test-UsingScope {
    # A variable in the functions local scope
    $Variable = 2
    Start-Job -ScriptBlock {
        $using:Global:Variable
    }
}
# A variable in global scope
$Variable = 1
Test-UsingScope | Receive-Job -Wait
```

The `$using` scope modifier acts in one direction only. Jobs can read from it, but never write to it. This limitation applies when attempting to write to collections as well as any attempts to create new variables:

```
PS> $hashtable = @{}
PS> Start-Job { $using:hashtable.Add('newValue', 1) } |
>>     Receive-Job -Wait
ParserError:
Line |
   1 |  Start-Job { $using:hashtable.Add('newValue', 1) } |
     |              ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     | Expression is not allowed in a Using expression.
```

Values returned from a job must use the output pipeline and be retrieved using `Receive-Job`.

The background operator

PowerShell 7 introduces the background operator feature. You can implement the background operator using an ampersand (`&`), which must be placed at the end of a statement. The placement of the operator is important. If placed at the beginning of a statement, `&` is the call operator. `&` must be the last thing in the statement; it cannot be used as an intermediate step in a pipeline, for example.

The following statement places the preceding statement into a job:

```
Get-Process &
```

The preceding command is therefore equivalent to the following `Start-Job` command:

```
Start-Job -ScriptBlock { Get-Process }
```

In both cases, the jobs are started in the current PowerShell working directory.

When using `Start-Job`, the working directory is implicitly inherited from the session starting the job.

When using the background operator, the working directory is explicitly set as the following shows:

```
PS> $job = Get-Process &
PS> $job.Command
Microsoft.PowerShell.Management\Set-Location -LiteralPath $using:pwd ; Get-Process
```

The preceding example shows that assignments can be used to capture the `job` object; they are not part of the background job.

The ThreadJob module

The `ThreadJob` module is shipped with PowerShell 7. `Start-ThreadJob` provides a faster alternative to `Start-Job`. It leverages PowerShell runspaces, which are explored at the end of this chapter to execute scripts.

You can use `Start-ThreadJob` in much the same way as `Start-Job`. Once a job has started, the remaining job commands

can be used to interact with it. For example, `Receive-Job` will be used to retrieve output:

```
Start-ThreadJob { Write-Host 'Hello world' } | Receive-Job -Wait
```

Help for the `ThreadJob` module is not available in PowerShell itself; the online reference must be used: https://docs.microsoft.com/powershell/module/threadjob/start-threadjob?view=powershell-7.1.

The `Start-ThreadJob` command allows a user to define an initialization script; this script runs in the runspace before the rest of the command starts. The initialization script cannot make use of the using scope modifier at this time. You can use this parameter to separate the components a job requires from the `job` script block itself.

Batching jobs

When working with the preceding commands, as well as runspaces, mentioned at the end of this chapter, it is important to consider what constitutes a job.

For example, if the task involves running a single command on 100 remote hosts, then there is little harm in allowing 100 to run at once. The system hosting the jobs only has to worry about connecting to the remote system and instructing it to get on with things.

If, on the other hand, the task is local and needs access to local resources, it may be more appropriate to run as many jobs as there are CPU cores on the host.

The same argument applies to jobs that have expensive setup steps, such as connecting to a remote service. It may be better to create batches of content rather than pushing everything into a job of its own.

Given an array of objects to process, you can create batches using a `for` loop:

```
$objects = foreach ($value in 1..1000) {
    [PSCustomObject]@{ Value = $value }
}
$batchSize = 100
$ScriptBlock = {
    # Long job set-up step
    Start-Sleep -Seconds 120
    foreach ($object in $using:batch) {
        # Perform action and create output
        $object
    }
}
for ($i = 0; $i -lt $objects.Count; $i += $batchSize) {
    $batch = $objects[$i..($i + $batchSize)]
    Start-Job -ScriptBlock $ScriptBlock
}
```

If each job is created using `$batch`, then each will act on 100 elements; a total of 10 jobs are created in this example.

Throttling might be added to ensure that the host is not overloaded by the request. Throttling techniques outside of the `ThrottleLimit` parameter were demonstrated earlier in this section.

Jobs provide a means of proactively executing code in an asynchronous manner. It is also possible to react to events.

Reacting to events

Events in .NET occur when something of interest happens to an object. For instance, `System.IO.FileSystemWatcher` can be used to monitor a filesystem for changes; when something changes, an event will be raised.

Many different types of objects raise events when changes occur. You can use `Get-Member` to explore an instance of an object for `Event` members. For example, a `Process` object returned by the `Get-Process` command includes several events, shown

as follows:

```
PS> Get-Process | Get-Member -MemberType Event
   TypeName: System.Diagnostics.Process
Name               MemberType Definition
----               ---------- ----------
Disposed           Event      System.EventHandler Disposed(System.Ob…
ErrorDataReceived  Event      System.Diagnostics.DataReceivedEventHa…
Exited             Event      System.EventHandler Exited(System.Obje…
OutputDataReceived Event      System.Diagnostics.DataReceivedEventHa…
```

PowerShell can react to these events, executing code when an event occurs.

This section uses the events raised by `FileSystemWatcher` to demonstrate working with events. `FileSystemWatcher` can react to a number of different events:

```
PS> [System.IO.FileSystemWatcher]::new() |
>>    Get-Member -MemberType Event |
>>    Select-Object Name
Name
----
Changed
Created
Deleted
Disposed
Error
Renamed
```

The following examples will use `Changed` and `Created`.

The Register-ObjectEvent and *-Event commands

`Register-ObjectEvent` is used to register interest in an event raised by a .NET object. The command creates a `PSEventSubscriber` object.

The `Register-ObjectEvent` command expects at least the name of the object that will be raising the event and the name of the event.

The following `FileSystemWatcher` instance watches the `C:\Data` folder. By default, the watcher will only watch for changes at that level; the `IncludeSubDirectories` property might be changed if this must change. Subscribers are created for the `Changed` and `Created` events in the following example:

```
$watcher = [System.IO.FileSystemWatcher]::new('C:\Data')
Register-ObjectEvent -InputObject $watcher -EventName Changed
Register-ObjectEvent -InputObject $watcher -EventName Created
```

If a file is created in the folder specified, an event will be raised. The `Get-Event` command can be used to view the event data:

```
PS> New-Item C:\Data\new.txt | Out-Null
PS> Get-Event
ComputerName     :
RunspaceId       : 46d2a562-2d07-4c58-9416-f82a3e9da5b8
EventIdentifier  : 3
Sender           : System.IO.FileSystemWatcher
SourceEventArgs  : System.IO.FileSystemEventArgs
SourceArgs       : {System.IO.FileSystemWatcher, new.txt}
SourceIdentifier : ff0784dc-1f0f-4214-b5e7-5d5516eaa13e
TimeGenerated    : 19/02/2019 17:29:53
MessageData      :
```

The `SourceEventArgs` property contains a `FileSystemEventArgs` object. This object includes the type of change, the path, and the filename.

The event remains until it is removed using `Remove-Event`, or the PowerShell session is closed. If another event is raised, it will be returned by `Get-Event` in addition to the existing event.

Depending on the operation performed, `FileSystemWatcher` may return more than one event. When using `Add-Content`, a single event will be raised as follows:

```
PS> Get-Event | Remove-Event
PS> Add-Content C:\Data\new.txt -Value value
PS> Get-Event | Select-Object -ExpandProperty SourceEventArgs
ChangeType    FullPath          Name
----------    --------          ----
   Changed    C:\Data\new.txt   new.txt
```

`Set-Content` is used when two events are raised. `Set-Content` makes two changes to the file, directly or indirectly. This will often be the case, depending on how an application interacts with the filesystem, which is shown as follows:

```
PS> Get-Event | Remove-Event
PS> Set-Content C:\Data\new.txt -Value value
PS> Get-Event | Select-Object -ExpandProperty SourceEventArgs
ChangeType    FullPath          Name
----------    --------          ----
   Changed    C:\Data\new.txt   new.txt
   Changed    C:\Data\new.txt   new.txt
```

Whether an event will trigger once or twice depends on the type in use, the event raised, and the subsystem that caused the event to be raised in the first place. Repeated events can potentially be ignored.

If events are being handled in the foreground using `Get-Event`, you can use `Wait-Event` to wait until an event is raised.

Wait-Event does not return any output

`Wait-Event` stops as soon as an event is raised. `Wait-Event` does not return the event; any raised events must be retrieved using `Get-Event`.

The Get-EventSubscriber and Unregister-Event commands

The `Get-EventSubscriber` command may be used to view any existing event handlers created using `Register-ObjectEvent`. For example, `Get-EventSubscriber` will display the subscribers created for `FileSystemWatcher`:

```
PS> Get-EventSubscriber
SubscriptionId   : 4
SourceObject     : System.IO.FileSystemWatcher
EventName        : Changed
SourceIdentifier : 6516aebc-d191-44b5-a38f-60314f606102
Action           :
HandlerDelegate  :
SupportEvent     : False
ForwardEvent     : False
SubscriptionId   : 5
SourceObject     : System.IO.FileSystemWatcher
EventName        : Created
SourceIdentifier : ff0784dc-1f0f-4214-b5e7-5d5516eaa13e
Action           :
HandlerDelegate  :
SupportEvent     : False
ForwardEvent     : False
```

If the subscribers are no longer required, they can be removed using the `Unregister-Event` command. The following command removes all registered event subscribers:

```
Get-EventSubscriber | Unregister-Event
```

The Action, Event, EventArgs, and MessageData parameters

The `Action` parameter of `Register-ObjectEvent` allows a script block to be automatically executed when an event is raised.

The script block can use a reserved variable, `$event`, which is equivalent to the output from `Get-Event`. In the following example, the event subscriber includes an action, which creates a log message. The log messages are written to file in a different folder; if they were written to the same folder, a loop would be created:

```
New-Item C:\Audit -ItemType Directory
$watcher = [System.IO.FileSystemWatcher]::new('C:\Data')
$params = @{
    InputObject = $watcher
    EventName   = 'Changed'
    Action      = {
        $event.SourceEventArgs |
            Export-Csv C:\Audit\DataActivity.log -Append
    }
}
Register-ObjectEvent @params
```

If a file is created in the `C:\Data` folder, an event will be raised, and an entry will be created in `C:\Audit\DataActivity.log`:

```
PS> Set-Content C:\Data\new.txt -Value new
PS> Import-Csv C:\Audit\DataActivity.log
ChangeType    FullPath           Name
----------    --------           ----
Changed       C:\Data\new.txt    new.txt
Changed       C:\Data\new.txt    new.txt
```

Additional information can be passed to the `Action` script block using the `MessageData` parameter. `MessageData` is an arbitrary object that contains user-defined information. Before continuing to the example, the event subscriber that was just created should be removed. The log file is also deleted as the format of the file will be changed:

```
Get-EventSubscriber | Unregister-Event
Remove-Item C:\Audit\DataActivity.log
```

The following example adds a date stamp to the log entry and a custom message, which is supplied via `MessageData`. The values passed in using the `MessageData` parameter are made available as a `MessageData` property on the `$event` variable:

```
$watcher = [System.IO.FileSystemWatcher]::new('C:\Data')
$params = @{
    InputObject = $watcher
    EventName   = 'Changed'
    Action      = {
        $user = $event.MessageData |
            Where-Object {
                $event.SourceEventArgs.Name -match $_.Expression
            } |
            Select-Object -ExpandProperty User -First 1
        $event.SourceEventArgs |
            Select-Object -Property @(
                @{Name = 'Date'; Expression = { Get-Date -Format u }}
                'ChangeType'
                'FullPath'
                @{Name = 'Responsible Person'; Expression = { $user }}
            ) |
            Export-Csv C:\Audit\DataActivity.log -Append
    }
    MessageData = @(
        [PSCustomObject]@{ Expression = '\.txt$'; User = 'Sarah' }
        [PSCustomObject]@{ Expression = '\.mdb';  User = 'Phil' }
    )
}
Register-ObjectEvent @params
```

Setting the content of a file in the `C:\Data` folder will trigger the event subscriber. An entry will be written to the log file using the entry from `MessageData`:

```
PS> Set-Content C:\Data\test.mdb 1
PS> Import-Csv C:\Audit\DataActivity.log
Date                  ChangeType  FullPath          Responsible Person
----                  ----------  --------          ------------------
2019-02-19 18:30:04Z  Changed     C:\Data\test.mdb  Phil
```

Event subscribers are globally scoped; they should be removed if they are no longer required. Closing the PowerShell session will remove all event subscribers.

Whether running jobs or reacting to events, the commands demonstrated in the previous sections are a great part of any PowerShell developer's toolkit. However useful they are, the commands lack fine control of the sessions they create and use. When greater control is required, PowerShell runspaces and runspace pools can be used directly via .NET types.

Using runspaces and runspace pools

Runspaces and runspace pools are an efficient way of asynchronously executing PowerShell code. Runspaces are far more efficient than jobs created by `Start-Job` as they execute in the same process. The main disadvantage is complexity: PowerShell does not include native commands to simplify working with these classes.

These days, the lack of native tooling is less of a problem. PowerShell 7 includes several alternatives that execute code in efficient runspaces, including `ForEach-Object` with the `-Parallel` parameter, and the `Start-ThreadJob` command.

In addition to these, the (now older) `PoshRSJob` module remains available on the PowerShell Gallery: https://www.powershellgallery.com/packages/PoshRSJob.

The `PoshRSJob` module is very mature and has a rich set of features. It was the most frequently recommended module, providing an alternative to the `Start-Job` command.

When more flexibility or efficiency is needed, it is helpful to understand how PowerShell can use runspaces directly.

Creating a PowerShell instance

`PowerShell` instances, runspaces in which PowerShell code can be executed, are created using the `Create` static method of the `System.Management.Automation.PowerShell` type. A type accelerator exists for this type and the name can be shortened:

```
$psInstance = [PowerShell]::Create()
```

> **System.Management.Automation.PowerShell or PowerShell**
>
> The usage is slightly confusing as both the console host and the type used here are normally referred to as PowerShell.
>
> References to instances of `System.Management.Automation.PowerShell` as `PowerShell` are highlighted in this section.

The object created by the `Create` method has a fluent interface. Methods can be chained one after another without assigning a value. The following example adds a single command and a parameter, and then runs the command:

```
[PowerShell]::Create().
    AddCommand('Get-Process').
    AddParameter('Name', 'powershell').
    Invoke()
```

A complex script can be built in this manner. If two commands are chained together, they are assumed to be part of the same statement, implementing a pipeline. The `AddStatement` method is used to start a new statement, ending the current command pipeline:

```
[PowerShell]::Create().
    AddCommand('Get-Process').AddParameter('Name', 'powershell').
    AddStatement().
    AddCommand('Get-Service').
    AddCommand('Select-Object').AddParameter('First', 1).
    Invoke()
```

The result of the preceding example is equivalent to the following script:

```
Get-Process -Name powershell
Get-Service | Select-Object -First 1
```

The `AddCommand`, `AddParameter`, and `AddStatement` methods demonstrated so far are particularly useful when assembling a script programmatically. If the script content is already known, the script can be added using the `AddScript` method:

```
$script = @'
    Get-Process -Name powershell
    Get-Service | Select-Object -First 1
'@
[PowerShell]::Create().AddScript($script).Invoke()
```

The script is added as a string, not as a script block. When creating the script, be mindful of variable expansion in double-quoted strings. Variable expansion is avoided in the previous example by enclosing the script content in single quotes.

The `AddScript` method can be used in conjunction with any of the other methods used here to build a complex set of commands.

The Invoke and BeginInvoke methods

The `Invoke` method used with each of the following examples executes the code immediately and synchronously. The

`BeginInvoke` method is used to execute asynchronously, that is, without waiting for the last operation to complete.

Both the `PowerShell` instance object and the `IASyncResult` returned by `BeginInvoke` must be captured. Assigning the values allows continued access to the instances and is required to retrieve output from the commands:

```
$psInstance = [PowerShell]::Create().
    AddCommand('Start-Sleep').AddParameter('Seconds', 300)
$asyncResult = $psInstance.BeginInvoke()
```

While the job is running, the `InvocationStateInfo` property of the `PowerShell` object will show as `Running`:

```
PS> $psInstance.InvocationStateInfo
State      Reason
-----      ------
Running
```

This state is reflected on the `IASyncResult` object held in the `$asyncResult` variable:

```
PS> $asyncResult | Format-List
CompletedSynchronously : False
IsCompleted            : False
AsyncState             :
AsyncWaitHandle        : System.Threading.ManualResetEvent
```

When the command completes, both objects will reflect that state:

```
PS> $psInstance.InvocationStateInfo.State
Completed
PS> $asyncResult.IsCompleted
True
```

Setting either (or both) of these variables to `null` does not stop the script from executing in the `PowerShell` instance. Doing so only removes the variables assigned, making it impossible to interact with the runspace:

```
$psInstance = [PowerShell]::Create().AddScript('
    1..60 | ForEach-Object {
        Add-Content -Path c:\temp\output.txt -Value $_
        Start-Sleep -Seconds 1
    }
')
$asyncResult = $psInstance.BeginInvoke()
$psInstance = $null
$asyncResult = $null
```

The script continues to execute, filling the output file. The following file may be using `Get-Content`:

```
Get-Content c:\temp\output.txt -Wait
```

If the work of the script is no longer required, the `Stop` method should be called instead of setting variables to null:

```
$psInstance = [PowerShell]::Create()
$psInstance.AddCommand('Start-Sleep').AddParameter('Seconds', 120).BeginInvoke()
$psInstance.Stop()
```

A terminating error is raised when the `Stop` method is called. If the output from the instance is retrieved using the `EndInvoke` method, a `The pipeline has been stopped` error message will be displayed.

The EndInvoke method and the PSDataCollection object

`EndInvoke` is one of two possible ways to get output from a `PowerShell` instance. The `EndInvoke` method may be called as follows:

```
$psInstance = [PowerShell]::Create()
$asyncResult = $psInstance.AddScript('1..10').BeginInvoke()
$psInstance.EndInvoke($asyncResult)
```

If the invocation has not finished, `EndInvoke` will block execution until it has completed.

The second possible method involves passing a `PSDataCollection` object to the `BeginInvoke` method:

```
$instanceInput = [System.Management.Automation.PSDataCollection[PSObject]]::new()
$instanceOutput = [System.Management.Automation.PSDataCollection[PSObject]]::new()
$psInstance = [PowerShell]::Create()
$asyncResult = $psInstance.AddScript('
    1..10 | ForEach-Object {
        Start-Sleep -Seconds 1
        $_
    }
').BeginInvoke(
    $instanceInput,
    $instanceOutput
)
```

The `$psInstance` and `$asyncResult` variables are still used to determine whether the script has completed. Results are available in `$instanceOutput` as they become available. Attempting to access `$instanceOutput` in the console will block execution until the script completes. New values added to the collection will be displayed as they are added.

The unused `$instanceInput` variable in the preceding example may be populated with values for an input pipeline if required, for example:

```
$instanceInput = [System.Management.Automation.PSDataCollection[PSObject]](1..10)
$instanceOutput = [System.Management.Automation.PSDataCollection[PSObject]]::new()
$psInstance = [PowerShell]::Create()
$asyncResult = $psInstance.
    AddCommand('ForEach-Object').
    AddParameter('Process', { $_ }).
    BeginInvoke(
        $instanceInput,
        $instanceOutput
    )
```

The `AddCommand` method was used in the preceding example as `ForEach-Object` will act on an input pipeline. A script can accept pipeline input within a process block; pipeline input is not implicitly passed to the commands within the script. The following example implements an input pipeline and uses the built-in `$_` variable to repeat the numbers from the input pipeline:

```
$instanceInput = [System.Management.Automation.PSDataCollection[PSObject]](1..10)
$instanceOutput = [System.Management.Automation.PSDataCollection[PSObject]]::new()
$asyncResult = $psInstance.AddScript('
    process {
        $_
    }
').BeginInvoke(
    $instanceInput,
    $instanceOutput
)
```

Each of the examples so far has concerned itself with running a single script or a set of commands.

Running multiple instances

As an individual instance is executing asynchronously with `BeginInvoke`, several may be started. In each case, both the `PowerShell` object and the `IASyncResult` object should be preserved:

```
$jobs = 1..5 | ForEach-Object {
    $instance = [PowerShell]::Create().AddScript('
        Start-Sleep -Seconds (Get-Random -Minimum 10 -Maximum 121)
    ')
    [PSCustomObject]@{
        Id          = $instance.InstanceId
        Instance    = $instance
        AsyncResult = $instance.BeginInvoke()
    } | Add-Member State -MemberType ScriptProperty -PassThru -Value {
        $this.Instance.InvocationStateInfo.State
    }
}
```

Each job will continue for a random number of seconds and then complete. As each job completes, the `State` property created by `Add-Member` will change to reflect that:

```
PS> $jobs | Select-Object Id, State
Id                                      State
--                                      -----
de79dcc3-8092-4592-a89e-271fc2b8b65e    Completed
85de5d4d-f754-461d-88da-ac5c7948c546    Running
eb8e0b84-2a47-4379-bd89-e7e523201033    Running
6357a4c3-b6d1-4a9f-8f88-ee3ac0891eb1    Running
3dc050fe-8ff9-4f93-afa9-86768bd3b407    Completed
```

The following snippet might be used to wait for all of the jobs to complete:

```
while ($jobs.State -eq 'Running') {
    Start-Sleep -Milliseconds 100
}
```

If the number of jobs is significantly larger, the system running the jobs might well become overwhelmed.

Using the RunspacePool object

`RunspacePool` can be used to overcome the problem of overwhelming a system. The pool can be configured with a minimum and maximum number of threads to execute at any point in time.

The `RunspacePool` object is created using the `RunspaceFactory` type, as follows:

```
[RunspaceFactory]::CreateRunspacePool(1, 5)
```

`RunspacePool` must be opened before it can be used. The same pool is set for each of the `PowerShell` instances that expects to use the pool:

```
$runspacePool = [RunspaceFactory]::CreateRunspacePool(1, 2)
$runspacePool.Open()
$jobs = 1..10 | ForEach-Object {
    $instance = [PowerShell]::Create().AddScript('Start-Sleep -Seconds 10')
    $instance.RunspacePool = $runspacePool
    [PSCustomObject]@{
        Id          = $instance.InstanceId
        Instance    = $instance
        AsyncResult = $instance.BeginInvoke()
    } | Add-Member State -MemberType ScriptProperty -PassThru -Value {
        $this.Instance.InvocationStateInfo.State
    }
}
```

Each of the jobs will show as running, but only two will complete at a time, based on the maximum set for the pool in the following example. After 10 seconds, the state of the jobs will be like the following:

```
PS> $jobs | Select-Object Id, State
Id                                      State
--                                      -----
63e2ab2d-613a-4c9c-8f21-d93c8a126008    Completed
781e4a08-04d6-4927-986a-e116fb16a852    Completed
1d80c45d-326b-423b-93d9-21703e747a93    Running
6840dfb1-f47d-4977-868f-697fcbb8af7e    Running
6f3aa668-f680-40b6-8a94-c9d04693b1ad    Running
868f324c-7ba5-4913-83a9-345d8f356aec    Running
318a44ec-b390-45a5-a2cc-0272c1e2ad20    Running
ced0f017-1a1c-42d0-9c53-9e09f9c8ace9    Running
9d003c91-6a2b-4d6f-820e-975fffeb57d8    Running
71818997-b55e-41d6-bdf2-e62426036863    Running
```

When all processing is finished, all objects should be explicitly disposed of to ensure they are closed:

```
$jobs.Instance | ForEach-Object Dispose
$runspacePool.Dispose()
```

After `Dispose` has been run, the variables might be set to `null`. Objects that are no longer referenced will be removed by garbage collection. Garbage collection can be run immediately using the following command if a large amount of memory was committed when running the jobs:

```
[GC]::Collect()
```

Runspace pools are incredibly useful. To improve the utility of the pool, it can be seeded with modules, functions, and variables before the pool is opened.

About the InitialSessionState object

`InitialSessionState` is used by `Runspace` or `RunspacePool` to describe a starting point. The `InitialSessionState` object may have modules, functions, or variables added.

PowerShell provides several different options for creating `InitialSessionState`. This is achieved using a set of static methods. The most used are `CreateDefault` and `CreateDefault2`. For example, `CreateDefault2` is used as follows:

```
$initialSessionState = [InitialSessionState]::CreateDefault2()
```

The difference between `CreateDefault` and `CreateDefault2` is that `CreateDefault` includes engine snap-ins, while `CreateDefault2` does not.

PowerShell Core does not use snap-ins

PowerShell Core does not include support for snap-ins. The difference between the two methods is therefore not apparent on PowerShell Core.

`CreateDefault2` is therefore slightly more lightweight and is more appropriate for more recent versions of PowerShell, that is, PowerShell 6 and greater.

In Windows PowerShell, the difference may be shown by creating and comparing the list of snap-ins in each case:

```
PS> [PowerShell]::Create([InitialSessionState]::CreateDefault()).AddCommand('Get-
PSSnapIn').Invoke().Name
Microsoft.PowerShell.Diagnostics
Microsoft.PowerShell.Host
Microsoft.PowerShell.Core
Microsoft.PowerShell.Utility
Microsoft.PowerShell.Management
Microsoft.PowerShell.Security
Microsoft.WSMan.Management
```

In Windows PowerShell, `CreateDefault2` only adds the `Microsoft.PowerShell.Core` snap-in, as follows:

```
PS> [PowerShell]::Create([InitialSessionState]::CreateDefault2()).
>>     AddCommand('Get-PSSnapIn').Invoke().Name
Microsoft.PowerShell.Core
```

Items can be added to `InitialSessionState` before `Runspace` (or `RunspacePool`) is opened.

Adding modules and snap-ins

You add modules using the `ImportPSModule` method of `InitialSessionState`:

```
$initialSessionState = [InitialSessionState]::CreateDefault2()
$initialSessionState.ImportPSModule('Pester')
```

Several modules can be added with the same method. Modules can be specified by name, in which case the most recent will be used. You can specify a module using a Hashtable that describes the name and version information:

```
$initialSessionState.ImportPSModule(@(
    'NetAdapter'
    @{ ModuleName = 'Pester'; ModuleVersion = '4.6.0' }
))
```

You can also use `MaximumVersion` and `RequiredVersion` with the Hashtable.

A snap-in may be imported in Windows PowerShell using the `ImportPSSnapIn` method. The method requires the name of a single snap-in, and a reference to a variable to hold any warnings raised during import:

```
$warning = [System.Management.Automation.Runspaces.PSSnapInException]::new()
$initialSessionState.ImportPSSnapIn('WDeploySnapin3.0', [Ref]$warning)
```

If multiple snap-ins are required, you must call the `ImportPSSnapIn` method once for each snap-in.

Adding variables

`InitialSessionState` objects created using `CreateDefault2` will include all of the built-in variables with default values. The value of these variables cannot be changed before the session is opened.

Additional variables can be added using the `Add` method of the `Variables` property. Variables are defined as a

`SessionStateVariableEntry` object. An example of adding a variable is shown here:

```
$variableEntry = [System.Management.Automation.Runspaces.SessionStateVariableEntry]::new(
    'Variable',
    'Value',
    'Optional description'
)
$initialSessionState = [InitialSessionState]::CreateDefault2()
$initialSessionState.Variables.Add($variableEntry)
```

Several overloads are available, each allowing the variable to be defined in greater detail. For example, a variable with the `Private` scope may be created:

```
$variableEntry = [System.Management.Automation.Runspaces.SessionStateVariableEntry]::new(
    'PrivateVariable',
    'Value',
    'Optional description',
    [System.Management.Automation.ScopedItemOptions]::Private
)
$initialSessionState.Variables.Add($variableEntry)
```

Defining a fixed type for a variable is more difficult; the `ArgumentTypeConverterAttribute` needed to do this is private and difficult to create in PowerShell. To work around this problem, you can create a variable with the required attributes, then `SessionStateVariableEntry` can be created from the variable:

```
[ValidateSet('Value1', 'Value2')][String]$ComplexVariable = 'Value1'
$variable = Get-Variable ComplexVariable
$variableEntry = [System.Management.Automation.Runspaces.SessionStateVariableEntry]::new(
    $variable.Name,
    $variable.Value,
    $variable.Description,
    $variable.Options,
    $variable.Attributes
)
$initialSessionState.Variables.Add($variableEntry)
```

Using this approach allows complex variables to be defined within the session.

Adding functions

You can add functions and other commands to the `InitialSessionState` object in much the same way as variables. If a function is within a module, the module should be imported instead.

Functions, as `SessionStateFunctionEntry` objects, are added to the `Commands` property of the `InitialSessionState` object.

Simple functions can be added by defining the body of the function inline, as follows:

```
$functionEntry = [System.Management.Automation.Runspaces.SessionStateFunctionEntry]::new(
    'Write-Greeting',
    'Write-Host "Hello world"'
)
$initialSessionState.Commands.Add($functionEntry)
```

You can add functions with scope options in the same way as with variables. Scoping is rarely used with functions.

If the function already exists in the current session, the output of `Get-Command` might be used to fill the `SessionStateFunctionEntry` object:

```
function Write-Greeting {
    Write-Host 'Hello world'
}
$function = Get-Command Write-Greeting
$functionEntry = [System.Management.Automation.Runspaces.SessionStateFunctionEntry]::new(
    $function.Name,
    $function.Definition
)
$initialSessionState.Commands.Add($functionEntry)
```

Once the `InitialSessionState` object is filled with the required objects, it may be used to create a `PowerShell` instance or a `RunspacePool`.

Using the InitialSessionState and RunspacePool objects

The `RunspacePool` object can be created using `RunspaceFactory`. `RunspacePool` can be created with either the minimum and maximum number of concurrent threads, or an `InitialSessionState` object.

Creating the pool using an `InitialSessionState` object is shown here:

```
$initialSessionState = [InitialSessionState]::CreateDefault2()
$runspacePool = [RunspaceFactory]::CreateRunspacePool($initialSessionState)
```

Any extra entries required in the `InitialSessionState` must either be added using the `$intialSessionState` variable before `RunspacePool` is created, or extra entries must be added using `$runspacePool.InitialSessionState` after `RunspacePool` is created. Changes cannot be made after `RunspacePool` has been opened.

If `RunspacePool` is created with `InitialSessionState`, the `SetMinRunspaces` and `SetMaxRunspaces` methods can be used to adjust the minimum and maximum number of threads. The default value for both the minimum and maximum is `1`. The following example changes the maximum:

```
$runspacePool.SetMaxRunspaces(5)
```

The `GetMinRunspaces` and `GetMaxRunspaces` methods may be used to retrieve the current values.

`RunspacePool` is then used as shown in the *Using the RunspacePool object* section.

Using Runspace-synchronized objects

Several classes in .NET offer runspace synchronization. This means that an instance of an object can be made accessible from runspaces that share a common parent.

The most commonly used runspace-synchronized object is a Hashtable. The Hashtable is created using the `Synchronized` static method of the `Hashtable` type:

```
$synchronizedHashtable = [Hashtable]::Synchronized(@{
    Key = 'Value'
})
```

The synchronized Hashtable can be added to an `InitialSessionState` object and then used within a script or command that is running in a runspace. The changes made to the Hashtable within the runspace are visible outside:

```
$variableEntry = [System.Management.Automation.Runspaces.SessionStateVariableEntry]::new(
    'synchronizedHashtable',
    $synchronizedHashtable,
    ''
)
$runspace = [RunspaceFactory]::CreateRunspace([InitialSessionState]::CreateDefault2())
$runspace.InitialSessionState.Variables.Add($variableEntry)
$psInstance = [PowerShell]::Create()
$psInstance.Runspace = $runspace
$runspace.Open()
$psInstance.AddScript(
    '$synchronizedHashtable.Add("NewKey", "NewValue")'
).Invoke()
```

After the script has completed, the key added by the script will be visible in the parent runspace in the current PowerShell session.

In addition to the runspace-synchronized Hashtable, an `ArrayList` might be created in a similar manner, as follows:

```
[System.Collections.ArrayList]::Synchronized(
    [System.Collections.ArrayList]::new()
)
```

.NET also offers classes in the `System.Collections.Concurrent` namespace, which offers similar cross-runspace access: https://docs.microsoft.com/dotnet/api/system.collections.concurrent.

For example, you can use `ConcurrentStack` as follows:

```
$stack = [System.Collections.Concurrent.ConcurrentStack[PSObject]]::new()
$stack.Push('Value')
$variableEntry = [System.Management.Automation.Runspaces.SessionStateVariableEntry]::new(
    'stack',
    $stack,
    ''
)
$runspace = [RunspaceFactory]::CreateRunspace([InitialSessionState]::CreateDefault2())
$runspace.InitialSessionState.Variables.Add($variableEntry)
$psInstance = [PowerShell]::Create()
$psInstance.Runspace = $runspace
$runspace.Open()
$psInstance.AddScript('
    $value = 0
    if ($stack.TryPop([Ref]$value)) {
        $value
    }
').Invoke()
```

Each of the collection types in the `System.Collections.Concurrent` namespace offers similar `Try` methods to access elements.

Summary

This chapter explored the job commands built into PowerShell; since they are built-in, they are a solid starting point for running asynchronous operations. Newer modules such as `ThreadJob` can be used to improve the efficiency of jobs.

You used event subscribers and commands to react to and work with events on .NET objects.

The final section detailed working with runspaces and runspace pools. These provide the greatest flexibility when working asynchronously.

16

Graphical User Interfaces

PowerShell is first and foremost a language built to work on the command line. Since PowerShell is based on .NET, it can use several different assemblies to create graphical user interfaces.

This chapter explores WPF, a common choice for writing graphical user interfaces in Windows. WPF is not cross-platform; the content of this chapter will only work on Windows.

Avalonia is a possible choice for a cross-platform framework, but the use of this is unfortunately beyond the capacity of this chapter: `http://avaloniaui.net/`.

The following topics are explored in this chapter:

- About Windows Presentation Foundation (WPF)
- Designing a UI
- About XAML
- Displaying the UI
- Layout
- Naming and locating elements
- Handling events
- Responsive interfaces

Windows Presentation Foundation is available in both Windows PowerShell and PowerShell 7, although the interfaces it creates can only be used on Windows.

About Windows Presentation Foundation (WPF)

Windows Presentation Foundation, or **WPF**, is a user interface framework. The components of a user interface are referred to as controls and include `Label`, `TextBox`, `Button`, and so on.

Before you can use WPF, you must load the `PresentationFoundation` assembly. The assembly can be loaded using `Add-Type`. This command is required once in each PowerShell session that intends to use WPF:

```
Add-Type -AssemblyName PresentationFramework
```

A WPF user interface typically defines all or most of the visible components in an **Extensible Application Markup Language** (**XAML**) document.

Designing a UI

It is difficult, but not impossible, to design a user interface in code alone. This chapter focuses on a small number of simple UI elements, which you can combine to build a more complex user interface. The examples in this chapter do not require a visual designer.

There are several options available for visual designers:

- Visual Studio – Free when using Community edition: `https://visualstudio.microsoft.com/vs/community/`
- PoshGUI – Web-based, requires a subscription: `https://poshgui.com/`
- PowerShell Pro Tools – Visual Studio Code extension, requires a subscription:

https://ironmansoftware.com/powershell-pro-tools-for-visual-studio-code/

In the case of Visual Studio, it can be used to generate the XAML content in the designer, and that XAML content can be reused in PowerShell.

About XAML

XAML is an XML document. XAML is used to describe the components or elements of a user interface. The following example describes a 350 by 350-pixel `Window` containing a `Label` :

```
<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <Label Content="Hello world" />
</Window>
```

The two namespace declarations in the `xmlns` and `xmlns:x` attributes are mandatory and cannot be omitted.

The document must first be read into an `XmlDocument` ; you can use the `Xml` type accelerator to do this. Then you must create a `XmlNodeReader` ; this can be created by casting an `XmlDocument` . Finally, the document is parsed using the `XamlReader` to create the user interface controls from the document:

```
$xaml = [xml]'<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <Label Content="Hello world" />
</Window>'
$window = [System.Windows.Markup.XamlReader]::Load(
    [System.Xml.XmlNodeReader]$xaml
)
```

The `$Window` variable contains the `Window` and all child elements it implements. The `Window` can be explored and changed in PowerShell before it is displayed, or it can be displayed immediately.

Displaying the UI

You can open the UI using the `ShowDialog` method of the `Window` :

```
$Window.ShowDialog()
```

In each of the examples in each of the following sections, the following short function can be used to view the `Window` :

```
function Show-Window {
    param (
        [Xml]$Xaml
    )
    Add-Type -AssemblyName PresentationFramework
    $Window = [System.Windows.Markup.XamlReader]::Load(
        [System.Xml.XmlNodeReader]$Xaml
    )
    $Window.ShowDialog()
}
```

Examples and Show-Window

The preceding function is reused in the examples that follow. If it is not present in the PowerShell session the examples will fail.

You can now use the function with the first XAML example:

```
$xaml = '<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <Label Content="Hello world" />
</Window>'
Show-Window $Xaml
```

If the first example has been added to the console it will display as follows:

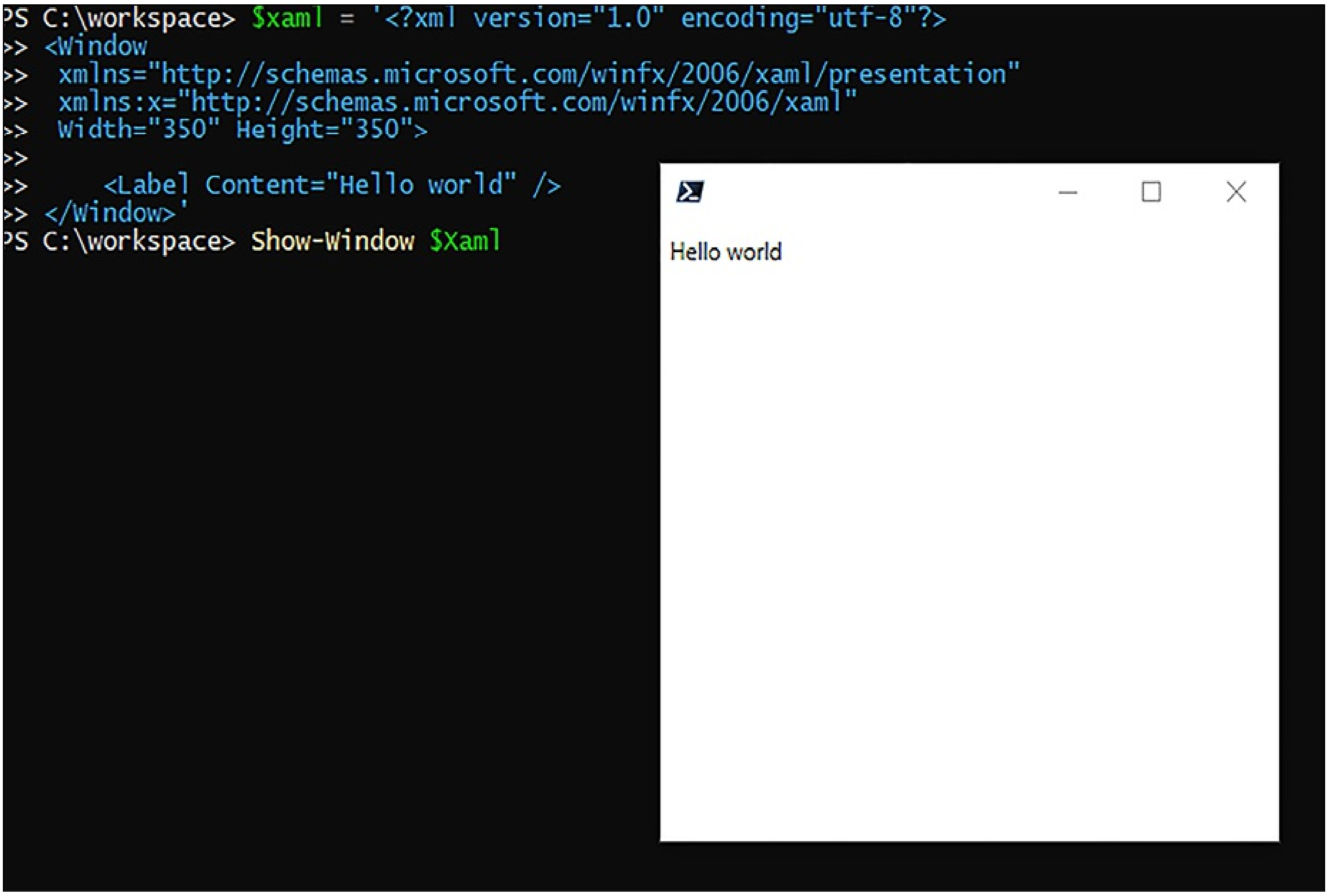


Figure 16.1: Showing the user interface

The properties and methods available for the `Window` are documented in Microsoft Docs: https://docs.microsoft.com/dotnet/api/system.windows.window.

In the preceding example, the `Window` contains a single `Label`. A `Window` can only directly contain a single child control, which is held in the `Content` property. The first few properties of the `Label` are shown here:

```
PS> $Window.Content
Target                      :
Content                     : Hello world
HasContent                  : True
ContentTemplate             :
ContentTemplateSelector     :
ContentStringFormat         :
BorderBrush                 :
BorderThickness             : 0,0,0,0
Background                  : #00FFFFFF
Foreground                  : #FF000000
```

WPF has a wide variety of layout controls that can be added to contain more controls. For example, a `Grid` control might be used to position `Label` and `TextBox` controls to create a simple form.

Layout

The layout of a WPF form is described by the elements it contains. Several controls are dedicated to positioning others. For example, a layout control may contain two `Label` controls.

Three of the different layout controls are explored:

- `Grid`
- `StackPanel`
- `DockPanel`

These three positioning controls can be used to create advanced layouts by specifying an absolute position for every single control by hand.

Layout controls such as those above can be nested inside one another to create more advanced interfaces.

A `Grid` can be used to arrange controls in rows and columns.

Using the Grid control

You can add a `Grid` control within the `Window` control in the `Xaml` document:

```
$xaml = '<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid>
        <Label Content="Hello world" />
    </Grid>
</Window>'
Show-Window $xaml
```

By default, `Grid` has one row and one column: a single cell. To add a second control to the grid, either the number of columns or the number of rows must be increased (or both). Rows and columns are numbered from `0` in the `Grid` control. Child controls are, by default, placed in row `0`, column `0`.

The following example adds a second column; the two controls are placed side by side by using `Grid.Row` and `Grid.Column` to explicitly define where each should appear:

```
$xaml = '<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Label Content="Row 1, Column 1"
         Grid.Row="0" Grid.Column="0" />
        <Label Content="Row 1, Column 2"
         Grid.Row="0" Grid.Column="1" />
    </Grid>
</Window>'
Show-Window $xaml
```

You can add rows to the grid in a similar manner. The next code snippet adds a second row to the grid:

```
$xaml = '<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <Label Content="Row 1, Column 1"
         Grid.Row="0" Grid.Column="0" />
        <Label Content="Row 1, Column 2"
         Grid.Row="0" Grid.Column="1" />
        <Label Content="Row 2, Column 1"
         Grid.Row="1" Grid.Column="0" />
        <Label Content="Row 2, Column 2"
         Grid.Row="1" Grid.Column="1" />
    </Grid>
</Window>'
Show-Window $xaml
```

The rows and columns in the grid are equally distributed. The `Width` and `Height` of each column or row can be changed by setting those properties in the `ColumnDefinition` or the `RowDefinition`.

The UI created by the preceding XAML document should look like this:

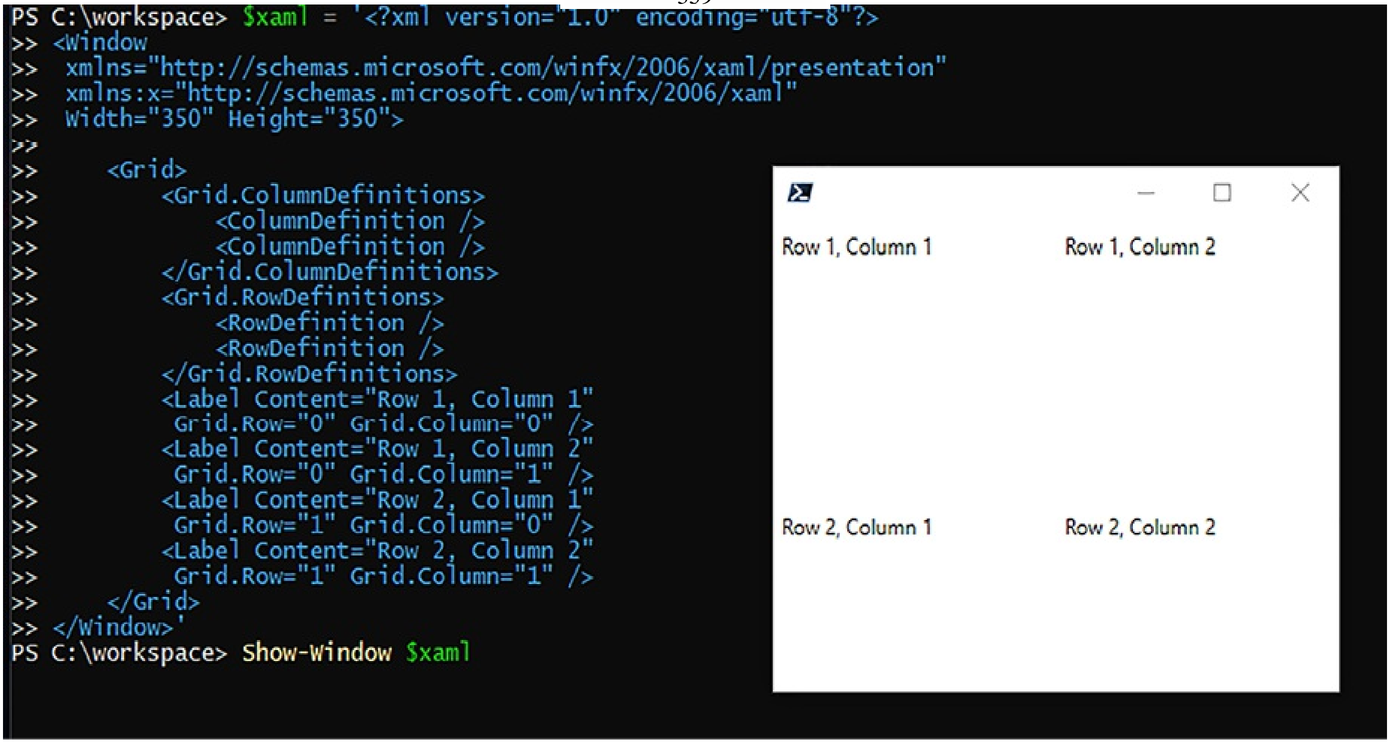


Figure 16.2: Grid with 2 columns and 2 rows

You can define column and row widths either in pixels or by fractional values based on the size of the parent (in this case, `Window`).

A fraction is described by following a numeric value with `*`. If `*` is used alone, the value is 1. In the following example, the first column takes three-quarters of the width, and the last column one-quarter. `ShowGridLines` has been enabled in the grid to more easily show the effect of the width control:

```
$xaml = '<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <Grid ShowGridLines="True">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="3*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <Label Content="Row 1, Column 1"
         Grid.Row="0" Grid.Column="0" />
        <Label Content="Row 1, Column 2"
         Grid.Row="0" Grid.Column="1" />
    </Grid>
</Window>'
Show-Window $xaml
```

This creates a window:

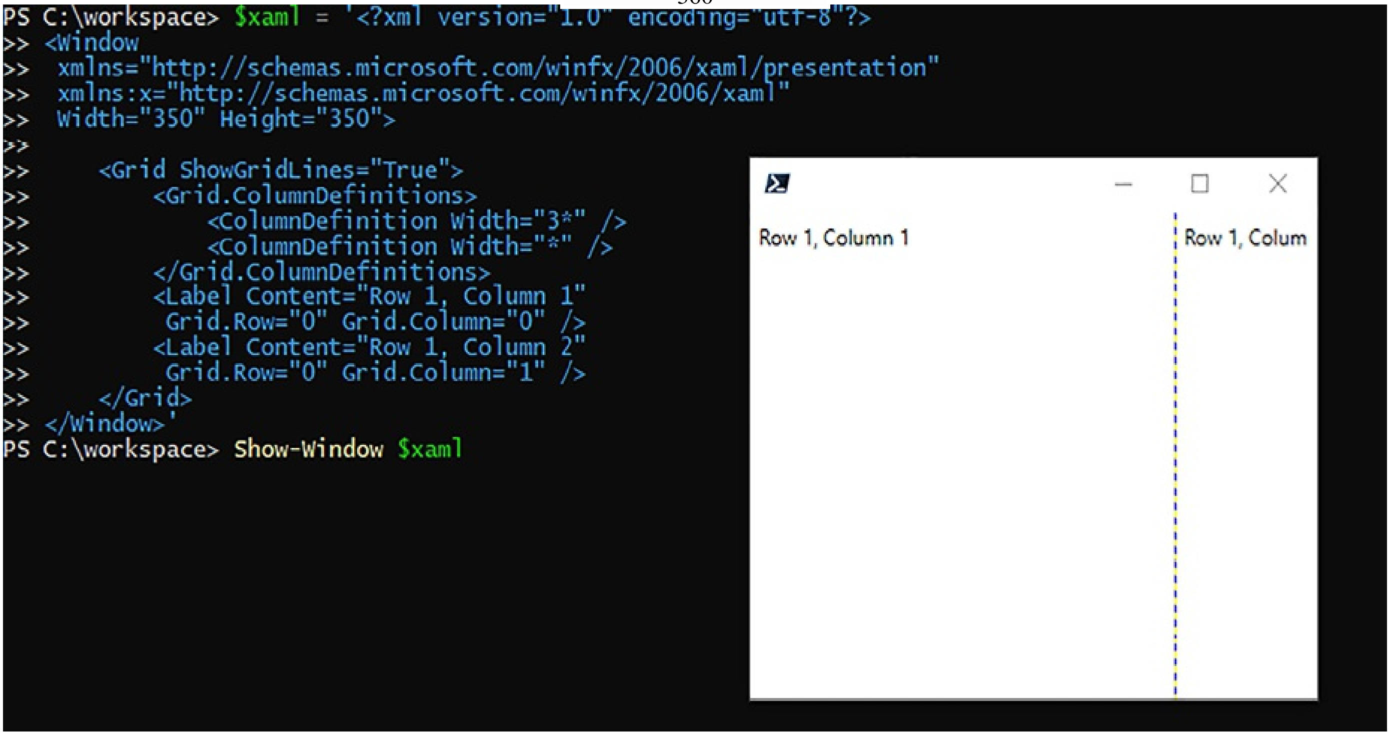


Figure 16.3: Grid control with fractional column width

The `Grid` control needs the number of rows and columns to be defined in advance. Each child control is placed within the grid using the `Grid.Row` and `Grid.Column` attributes as shown in the preceding examples.

A `StackPanel` can be used to arrange controls in a vertical or horizontal stack.

Using the StackPanel control

A `StackPanel` control allows one control to be placed next to another. The `StackPanel` can be arranged horizontally or vertically. Controls can be added from left to right, or right to left. When the orientation is vertical, left to right is equivalent to top to bottom.

A `StackPanel` is an ideal choice for a list of controls, such as a list of buttons. In the following example, the buttons are arrays vertically (the default) within the `StackPanel`:

```
$xaml = '<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <StackPanel>
        <Button Content="Button1" />
        <Button Content="Button2" />
        <Button Content="Button3" />
    </StackPanel>
</Window>'
Show-Window $xaml
```

The orientation can be changed with an attribute on the `StackPanel` element in the XAML document:

```
$xaml = '<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <StackPanel Orientation="Horizontal">
        <Button Content="Button1" />
        <Button Content="Button2" />
        <Button Content="Button3" />
    </StackPanel>
</Window>'
Show-Window $xaml
```

If a `Height` or `Width` is set, the `StackPanel` will limit the size of the elements it contains. In the following example, the `Width` of the `StackPanel` is reduced to 50 pixels; it now only takes up a small part of the `Window`:

```
$xaml = '<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <StackPanel Width="50">
        <Button Content="Button1" />
        <Button Content="Button2" />
        <Button Content="Button3" />
    </StackPanel>
</Window>'
Show-Window $xaml
```

The result of setting the width is shown next. As there is no code behind this user interface, pressing the buttons will have no effect:

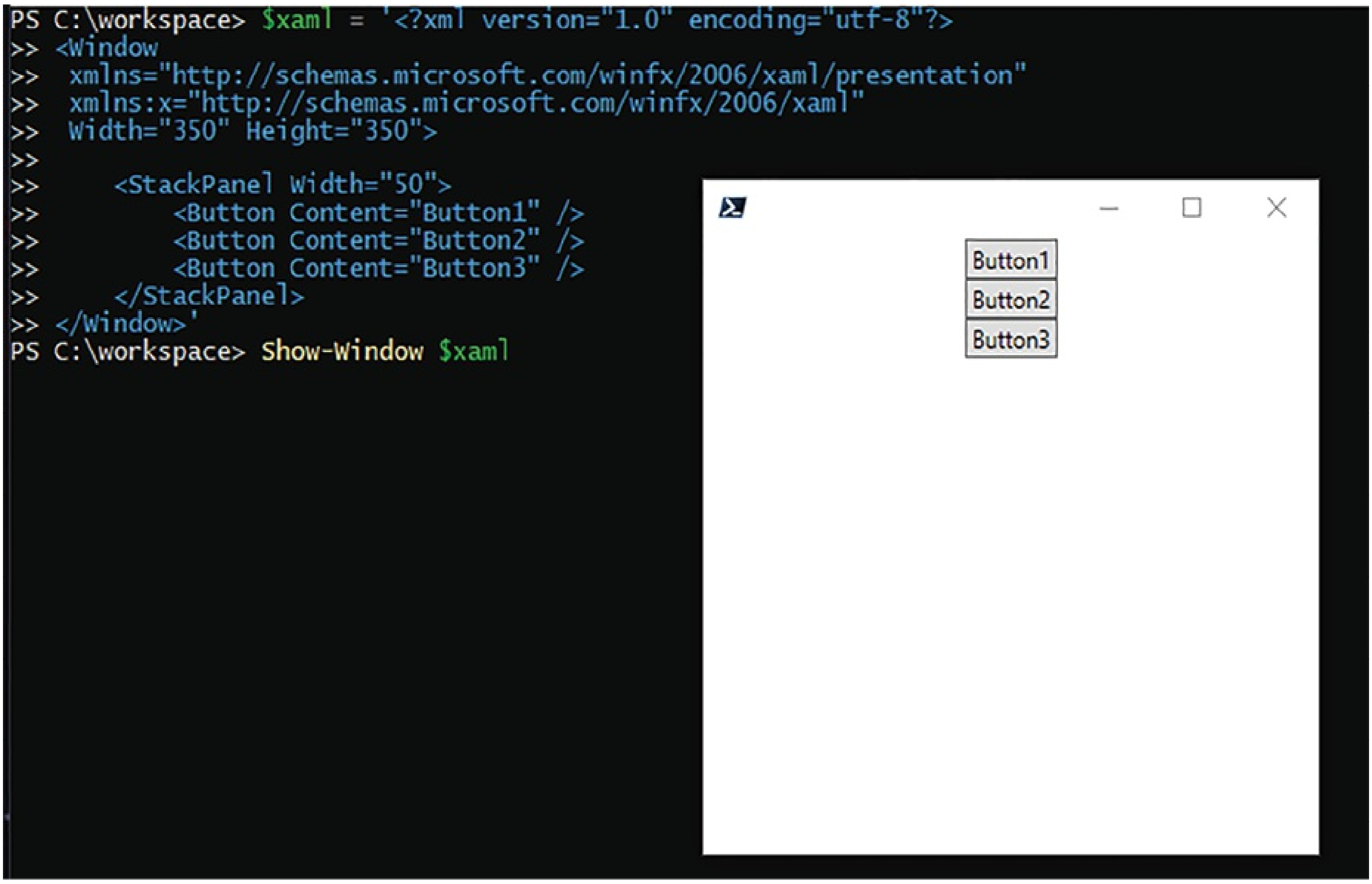


Figure 16.4: StackPanel positioned in the top center of a window

Elements within the `StackPanel` cannot exceed the `Width` and `Height` of that `StackPanel`.

The `StackPanel` is not especially useful where it is; the buttons are in the middle of the `Window`, which makes it hard to place more controls (`Label`, `TextBox`, `Button`, and so on). The `StackPanel` would be better placed within another control. A `Grid` is one option, but it is potentially difficult to arrange differently sized elements within a `Grid`. A `DockPanel` offers an easier alternative for putting together controls of different sizes.

Using the DockPanel control

A `DockPanel` is used to arrange elements around the edge of a rectangle. Each element is positioned using the `DockPanel.Dock` attribute, and the possible values are `Top`, `Bottom`, `Left`, and `Right`.

By default, the last control in the `DockPanel` uses up the rest of the available space. The `StackPanel` with the list of buttons might be positioned within a `DockPanel`, perhaps aligning it to the left, and a `Label` is added as a placeholder after this to use up the remaining space:

```
$xaml = '<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <DockPanel>
        <StackPanel Width="50" DockPanel.Dock="Left">
            <Button Content="Button1" />
            <Button Content="Button2" />
            <Button Content="Button3" />
        </StackPanel>
        <Label />
    </DockPanel>
</Window>'
Show-Window $xaml
```

The outcome is shown here:

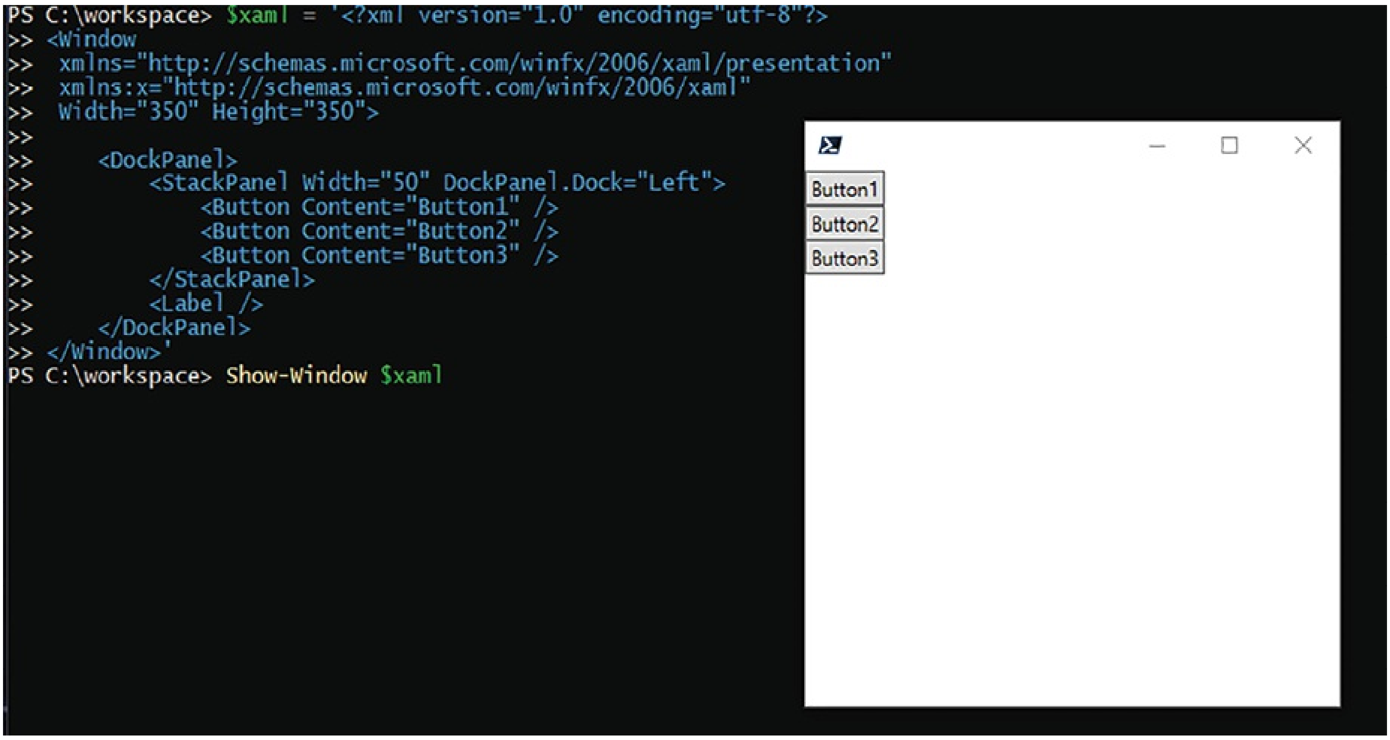


Figure 16.5: DockPanel and StackPanel

`DockPanel` allows elements to be arranged like a sliding puzzle. The following example includes each of the following elements:

1. A `Label` at the top

2. A `StackPanel` to the left
3. A `ComboBox` at the top
4. A `StackPanel` at the bottom
5. A `Label` to use up all remaining space

The order of the elements determines how much of the remaining space is available:

```
$xaml = '<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="800" Height="500">
    <DockPanel>
        <Label Height="50" Background="Gainsboro"
         DockPanel.Dock="Top" />
        <StackPanel Background="LightBlue" Width="250"
         DockPanel.Dock="Left" />
        <ComboBox Margin="5" DockPanel.Dock="Top" />
        <StackPanel Height="30"
         Orientation="Horizontal"
         FlowDirection="RightToLeft"
         DockPanel.Dock="Bottom">
            <Button Content="Exit" Width="50" Margin="5" />
            <Button Content="Cancel" Width="50" Margin="5" />
            <Button Content="OK" Width="50" Margin="5" />
        </StackPanel>
        <Label Background="LightCoral" DockPanel.Dock="Top" />
    </DockPanel>
</Window>'
Show-Window $xaml
```

The top-most `Label` and `StackPanel` have a background color set to show how each of the elements takes up space. The final `Label` is also colored, highlighting the unused space in the interface.

The resulting interface is shown in *Figure 16.6*. As before, the user interface has no code behind it, so none of the buttons work:

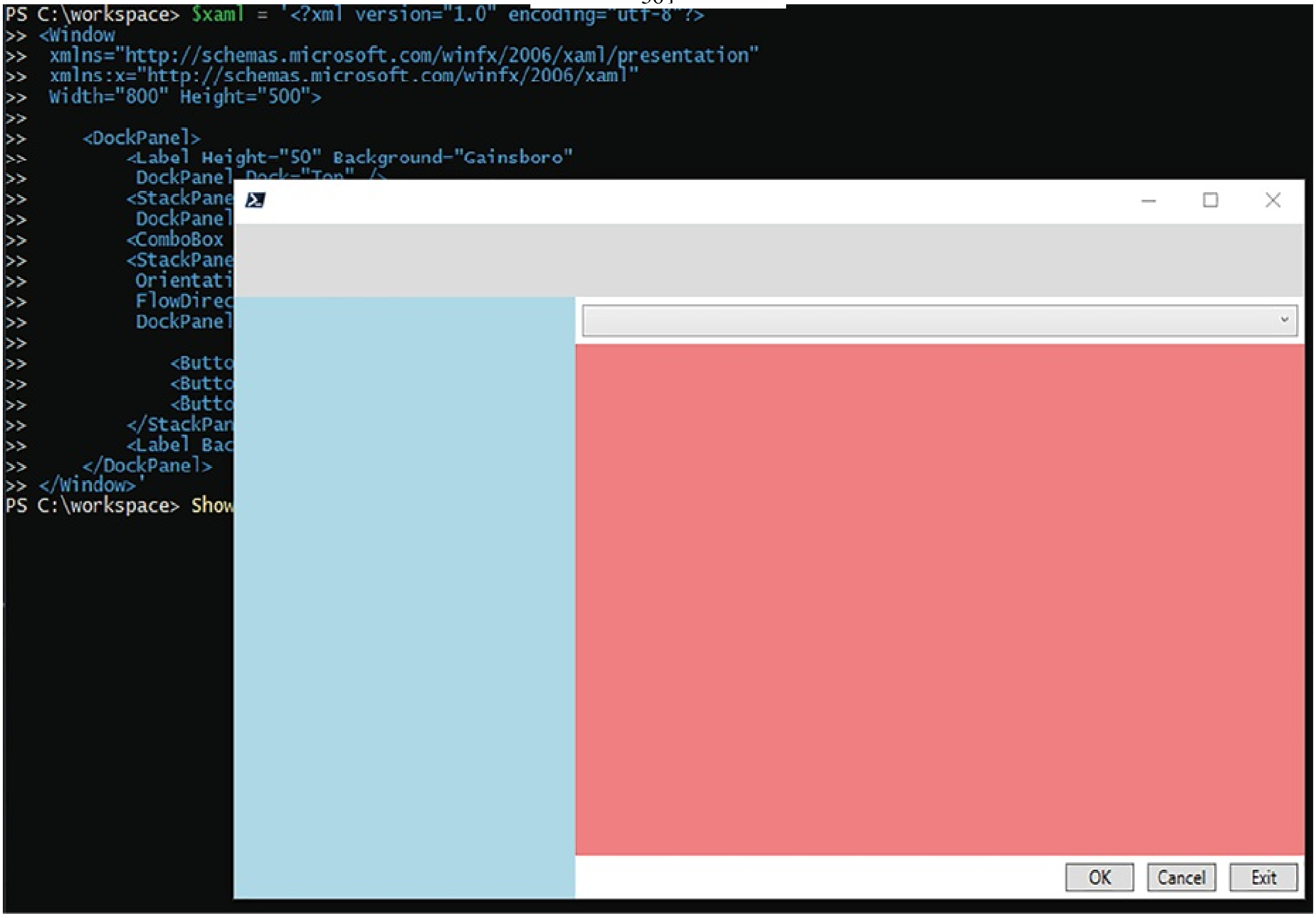


Figure 16.6: Multi-element docking

Each element of the user interface might be filled with different content.

The preceding example also makes use of several other layout properties, such as `Margin` and `Padding`.

About Margin and Padding

The `Margin` of a control is the space around the outside of that control. Most controls can use the `Margin` property. Notable exceptions include the `RowDefinition` and `ColumnDefinition` types. Margins for elements in a Grid must be defined on each nested element.

`Margin` and `Padding` are often used together to space out a user interface, preventing elements from running into each other, which results in a crowded user interface.

`Padding` is the space between the edge of a control and the `Content`. Controls like `Button`, `Label`, `TextBox`, and `ComboBox` all have a `Padding` property.

Both `Padding` and `Margin` can be defined on `Left`, `Right`, `Top`, and `Bottom`. Ultimately this creates a `System.Windows.Thickness` object after the XAML has been read into PowerShell.

If a single value is used, as is the case with the `ComboBox` control used in the last example, the value will be applied to all four margins. The XAML element is shown here:

```
<ComboBox Margin="5" DockPanel.Dock="Top" />
```

It is possible to see the impact of this value by casting the `Margin` value to the `Thickness` type in PowerShell:

```
PS> [System.Windows.Thickness]'5'
Left Top Right Bottom
---- --- ----- ------
   5   5     5      5
```

The `Thickness` type is only available if the `Add-Type` command to load the `PresentationFoundation` assembly at the beginning of this section has been run in the PowerShell session.

If a control requires different margins, a comma-separated list of values can be set. For instance, if `Margin` is set as shown below then only the `Top` and `Bottom` margins will be set:

```
<ComboBox Margin="0,5,0,5" DockPanel.Dock="Top" />
```

As before, you can test the effect of these values by casting the comma-separated string to the `Thickness` type:

```
PS> [System.Windows.Thickness]'0,5,0,5'
Left Top Right Bottom
---- --- ----- ------
   0   5     0      5
```

The same approach used with `Margin` can be used with `Padding` values.

The following example attempts to show the impact of setting a `Margin` and `Padding` in a control:

```
$xaml = '<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <Grid ShowGridLines="True">
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <TextBox Text="No margin, no padding"
         Grid.Row="0" />
        <TextBox Text="Margin, no padding" Margin="5"
         Grid.Row="1" />
        <TextBox Text="Padding, no margin" Padding="5"
         Grid.Row="2" />
        <TextBox Text="Padding and margin" Padding="5" Margin="5"
         Grid.Row="3" />
    </Grid>
</Window>'
Show-Window $xaml
```

A `Grid` is displayed with grid lines to show the boundary of each cell, and the `TextBox` is used as it has a border. The `Margin` is shown to affect the space around the outside of the `TextBox`; between the `Grid` cell edge and the `TextBox` edge. `Padding` is shown to affect the space between the `TextBox` edge and the text within the `TextBox`:

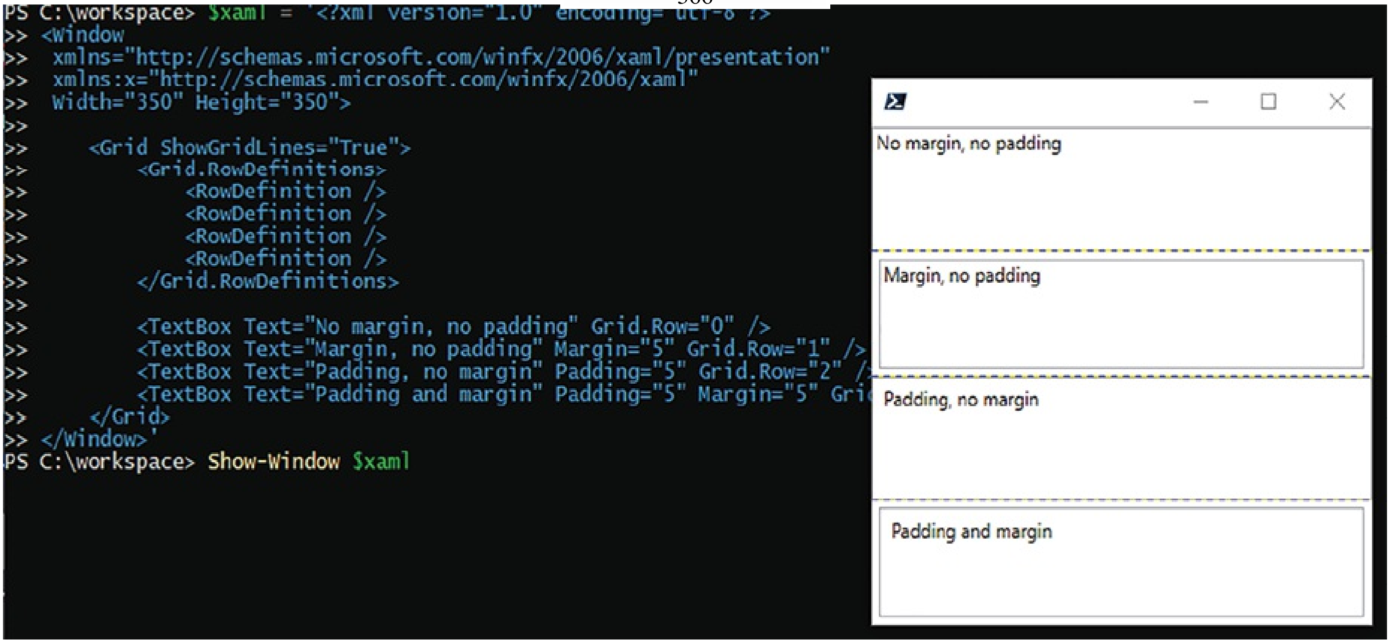


Figure 16.7: Margins and padding

The elements in the preceding XAML documents are only accessible by working down through the properties from the `$Window` variable.

Naming and locating elements

It is only possible to perform actions on elements in the UI if they can be located. None of the controls in the preceding examples have been given names. Locating one of the buttons in a `StackPanel`, for example, is difficult as it stands.

Returning to the `StackPanel` example, it has three buttons:

```
$xaml = [xml]'<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <StackPanel Width="50">
        <Button Content="Button1" />
        <Button Content="Button2" />
        <Button Content="Button3" />
    </StackPanel>
</Window>'
$window = [System.Windows.Markup.XamlReader]::Load(
    [System.Xml.XmlNodeReader]$xaml
)
```

Accessing the `Text` property of `Button3` as it stands requires the following relatively complex statement (in comparison to the complexity of the UI):

```
$button3 = $window.Content.Children[2].Content
```

This statement will potentially break if another button is added; it can therefore be said to be a bad practice.

Instead of locating controls by position, a control can be given a name, and the name used to find that control in the UI:

```
$xaml = [xml]'<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <StackPanel Width="50">
        <Button Name="Button1" Content="Button1" />
        <Button Name="Button2" Content="Button2" />
        <Button Name="Button3" Content="Button3" />
    </StackPanel>
</Window>'
$window = [System.Windows.Markup.XamlReader]::Load(
    [System.Xml.XmlNodeReader]$xaml
)
```

Now it has a name, each button (or any other named control) can be located by using a `FindName` method on any other element in the UI. For example:

```
PS> $button1 = $window.FindName('Button1')
PS> $button1.Name
Button1
PS> $button1.FindName('Button2').Name
Button2
```

`FindName`, when used in an event handler, can create difficult-to-resolve scoping problems in PowerShell; properties that are expected to be set may appear to be empty. Instead, each of the named controls can be gathered into a Hashtable so each is easily accessible wherever it might be needed:

```
$xaml = [xml]'<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <StackPanel Width="50">
        <Button Name="Button1" Content="Button1" />
        <Button Name="Button2" Content="Button2" />
        <Button Name="Button3" Content="Button3" />
    </StackPanel>
</Window>'
$window = [System.Windows.Markup.XamlReader]::Load(
    [System.Xml.XmlNodeReader]$xaml
)
$controls =  @{}
foreach ($control in $xaml.SelectNodes('//*[@Name]')) {
    $controls[$control.Name] = $Window.FindName($control.Name)
}
$controls['Button1']
```

The preceding `SelectNodes` method (used on the `XmlDocument`) finds all the elements in the XML document that have been given a name. It is therefore important to ensure that all names uniquely identify a single control.

You can create a short function to provide access to the top-level `Window` and all named child controls:

```
function Import-Xaml {
    param (
        [Xml]$Xaml
    )
    Add-Type -AssemblyName PresentationFramework
    $window = [System.Windows.Markup.XamlReader]::Load(
        [System.Xml.XmlNodeReader]$Xaml
    )
    $controls = @{}
    foreach ($control in $Xaml.SelectNodes('//*[@Name]')) {
        $controls[$control.Name] = $window.FindName($control.Name)
    }
    [PSCustomObject]@{
        MainWindow = $Window
        Controls   = $controls
    }
}
```

Examples and Import-Xaml

The preceding function is reused in the examples that follow. If it is not present in the PowerShell session, the examples will fail.

The preceding function is shown below with the example used earlier in this section:

```
$xaml = '<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <StackPanel Width="50">
        <Button Name="Button1" Content="Button1" />
        <Button Name="Button2" Content="Button2" />
        <Button Name="Button3" Content="Button3" />
    </StackPanel>
</Window>'
$ui = Import-Xaml $Xaml
```

The `Controls` property contains the three buttons used in the interface:

```
PS> $ui.Controls
Name                       Value
----                       -----
Button2                    System.Windows.Controls.Button: Button2
Button1                    System.Windows.Controls.Button: Button1
Button3                    System.Windows.Controls.Button: Button3
```

The `MainWindow` property can be used to access the `Window` itself and show the UI:

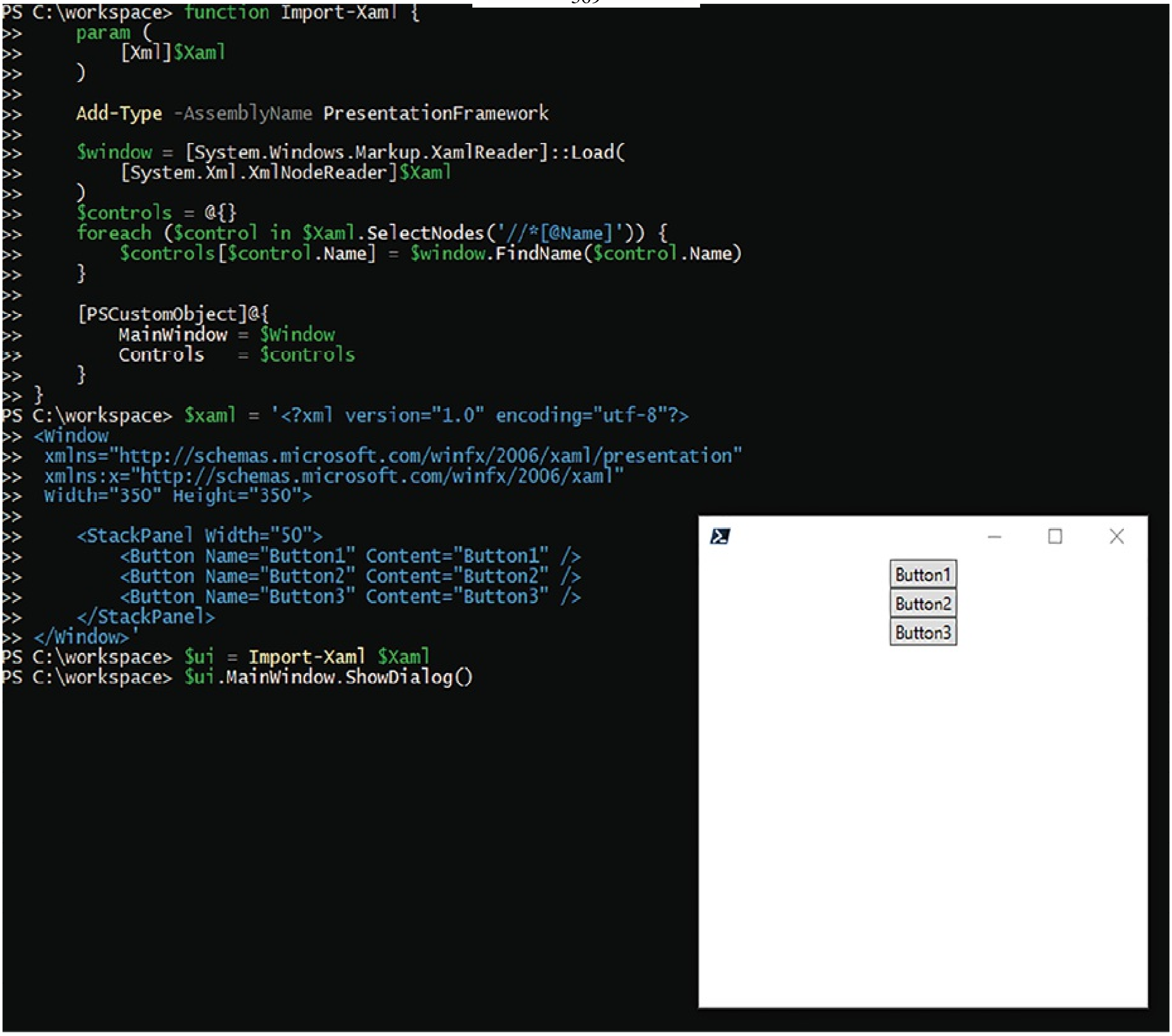


Figure 16.8: Using Import-Xaml

The ability to locate individual elements in code allows changes to be made to individual elements, which is essential when adding working with event handlers.

Handling events

An event is raised in a UI when a button is pressed, when a selection changes, when a key is pressed, and so on.

To react to an event, an event handler must be created and attached to the control that raises the event.

Event handlers must be added before `ShowDialog()` is run.

The list of possible events for the `Window` is extensive. Microsoft Docs lists the events and briefly describes each: https://docs.microsoft.com//dotnet/api/system.windows.window#events.

One possible event is pressing the Escape key while the UI has focus. It might be desirable to close the UI in this case. This is the `KeyDown` event and can be attached to the `Window` object.

For example, a `KeyDown` handler can be added to the following UI:

```
$xaml = '<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <Label Content="Hello world" />
</Window>'
$ui = Import-Xaml $xaml
```

Event handlers are added by running a method named after the event. The method name is prefixed with `add_`. `Get-Member` can be used to show the methods, but the `-Force` parameter is required. For example:

```
PS> $ui.MainWindow | Get-Member add_KeyDown -Force
   TypeName: System.Windows.Window
Name        MemberType Definition
----        ---------- ----------
add_KeyDown Method     void add_KeyDown(System.Windows.Input.K...
```

Two arguments are automatically made available to every event handler: the control that raised the event (`$sender`) and any event arguments (`$eventArgs`).

In the case of the `KeyDown` event, the `$sender` is the `Window` object, and `$eventArgs` includes the `Key` property describing which key was pressed.

Therefore, an event handler that allows the Escape key (`ESC`) to close the window can be added:

```
$ui.MainWindow.add_KeyDown({
    param ( $sender, $eventArgs )
    if ($eventArgs.Key -eq 'ESC') {
        $sender.Close()
    }
})
```

Adding the event handler alone like this has no visual impact. The change is only relevant when the UI is started.

With this event handler added, pressing Escape at any time after the UI has opened and with the UI having focus will close the UI:

```
$ui.MainWindow.ShowDialog()
```

This event handler is easy to implement as the only object it needs to act on is the `Window` itself. The `Window` is available in both the `$Window` variable and the `$sender` parameter; either variable can be used to run the `Close` method.

A similar approach can be taken to make an `Exit` button work.

Buttons and the Click event

The `Click` event on buttons can be used by adding an event handler. In the following example, the `Exit` button is used to close the parent window. First a UI is defined with an `Exit` button, and the button is given a name:

```
$xaml = '<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <Button Name="Button" Content="Exit" />
</Window>'
$ui = Import-Xaml $xaml
```

Once created, the event handler for `Click` is added to the `Button`:

```
$ui.Controls['Button'].add_Click({
    param ( $sender, $eventArgs )
    $ui.MainWindow.Close()
})
$ui.MainWindow.ShowDialog()
```

The controls in the UI are subject to default styling; moving the mouse over the button, for example, will cause the button to be highlighted. A detailed exploration of styling is beyond the scope of this chapter.

Controls in the UI can also read from and write to other controls.

In the following example, the `Click` event for the button is changed to update the content of another `Label`:

```
$xaml = '<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <StackPanel>
        <Button Name="Button" Content="Run" />
        <Label Name="Label" />
    </StackPanel>
</Window>'
$ui = Import-Xaml $xaml
$ui.Controls['Button'].add_Click({
    param ( $sender, $eventArgs )
    $ui.Controls['Label'].Content = 'Hello world'
})
$ui.MainWindow.ShowDialog()
```

In the preceding example, the `StackPanel` is the only element without a name. In the case of this particular UI, it does not need a name as it is not referenced by any of the code.

The result of pressing the button in the UI is shown in the following example. Before the button is pushed, the area below the button will contain a blank `Label`:

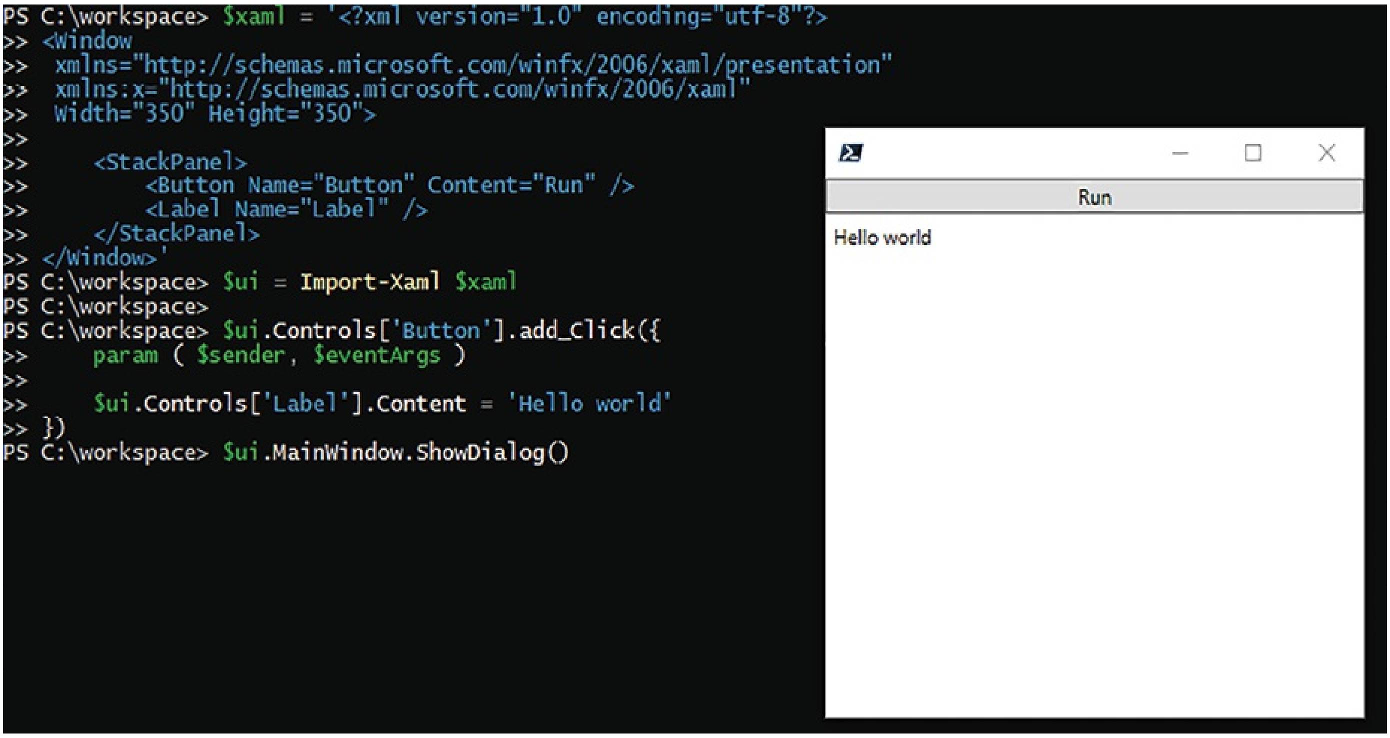


Figure 16.9: UI after pressing the Run button

Like `Button`, a `ComboBox` has specific events associated with the possible behavior of that control.

ComboBox and SelectionChanged

`SelectionChanged` can be used in a similar way to the `Click` event on a button. When the selection changes, the selection is available via the `SelectedItem` property of the `ComboBox` control.

The value in `SelectedItem` is a `ComboBoxItem` type; the actual value selected is held in the `Content` property of the item:

```
$xaml = '<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <StackPanel>
        <ComboBox Name="ComboBox">
            <ComboBoxItem>Apple</ComboBoxItem>
            <ComboBoxItem>Orange</ComboBoxItem>
        </ComboBox>
        <Label Name="Label" />
    </StackPanel>
</Window>'
$ui = Import-Xaml $xaml
$ui.Controls['ComboBox'].add_SelectionChanged({
    param ( $sender, $eventArgs )
    $ui.Controls['Label'].Content = (
        'Selected a {0}' -f $sender.SelectedItem.Content
    )
})
$ui.MainWindow.ShowDialog()
```

Each control has different events available. Microsoft Docs and `Get-Member` can be used to explore the capabilities of that control.

Each of the values above has set a simple text value in the UI. If the content retrieved is more complex, dynamically creating controls to describe the output might be desirable.

Adding elements programmatically

One control might add more values to the UI when an event is triggered. For example, clicking a button might cause the results of a command to be displayed.

The previous examples can already deal with simple text output, but PowerShell rarely returns a simple string as the output from a command. You can use a `ListView` control to display more complex values.

The following `Window` is the starting point for this example:

```
$xaml = '<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <DockPanel>
        <Button Name="Button" Content="Get-Process"
         DockPanel.Dock="Top" />
        <ListView Name="ListView">
            <ListView.View>
                <GridView />
            </ListView.View>
        </ListView>
    </DockPanel>
</Window>'
$ui = Import-Xaml $xaml
```

The UI includes a button that will run a command. The `Click` event for the button updates the columns and items in the `ListView`:

```
$ui.Controls['Button'].add_Click({
    param ( $sender, $eventArgs )
    $data = Get-Process | Select-Object Name, ID
    $listView = $ui.Controls['ListView']
    # Clear any previous content
    $listView.View.Columns.Clear()
    foreach ($property in $data[0].PSObject.Properties) {
        $column = [System.Windows.Controls.GridViewColumn]@{
            DisplayMemberBinding = (
                [System.Windows.Data.Binding]$property.Name
            )
            Header               = $property.Name
        }
        $listView.View.Columns.Add($column)
    }
    $listView.ItemsSource = $data
})
$ui.MainWindow.ShowDialog()
```

When the UI is opened, and the button is clicked, the `Name` and `ID` of each running process will be displayed in the `ListView`.

The hidden member, `PSObject`, is used to dynamically discover the properties of whatever objects are held in the `$data` variable. In this case, those will only be `Name` and `ID` as `Select-Object` was used. The filtered output from the preceding example should look like this:

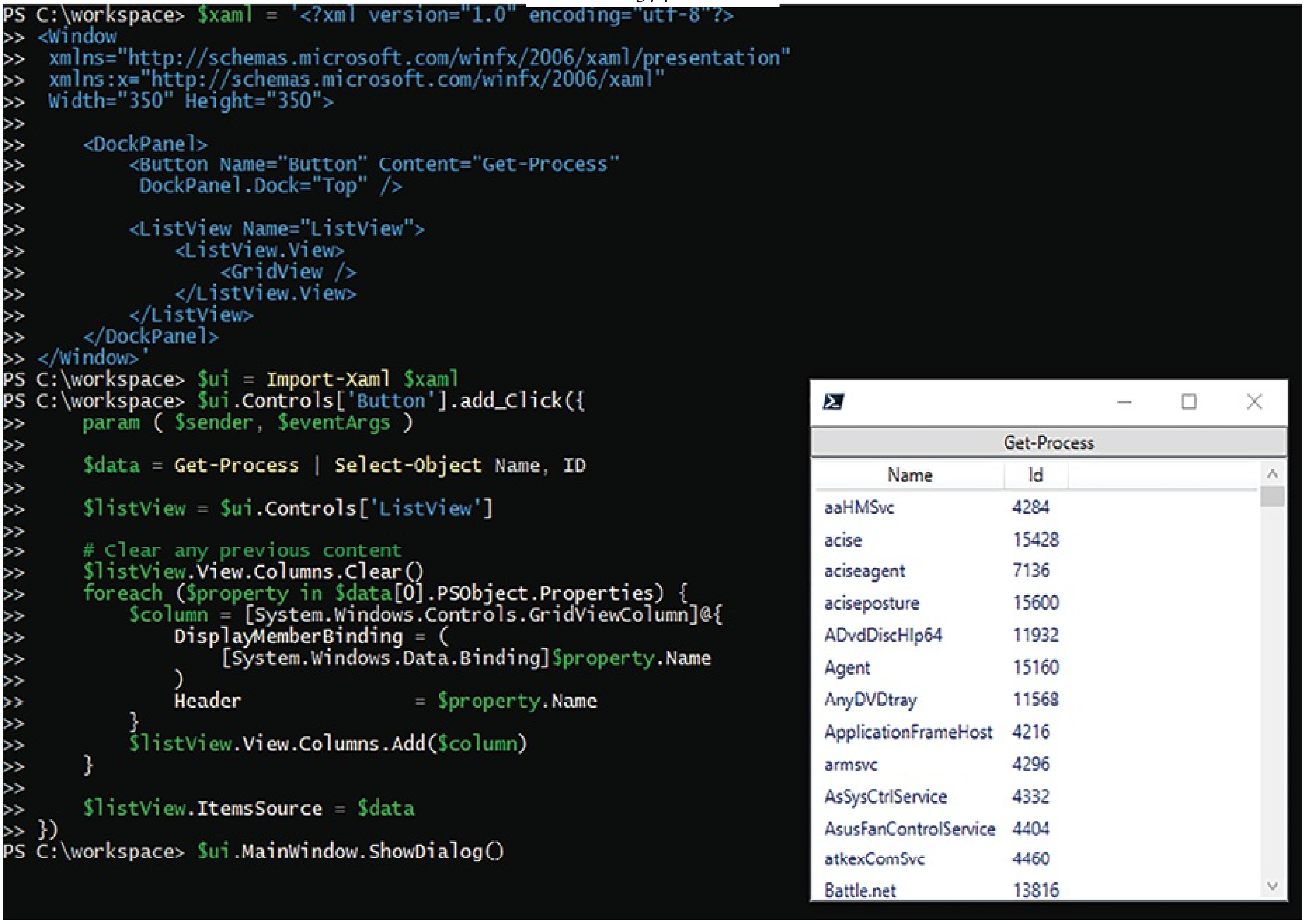


Figure 16.10: Updating ListView

The dynamically added content might be extended further to include an event handler that acts when a header is clicked.

Sorting a ListView

Each of the columns in a `ListView` can be sorted by adding a `Click` handler to the column header.

The sorting handler is relatively complex; it should account for any existing sorting if it is to sort in either direction:

To begin, a UI is created with a `ListView`:

```
$xaml = '<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="450" Height="800">
    <DockPanel>
        <Button Name="Button" Content="Get-Process"
         DockPanel.Dock="Top" />
        <ListView Name="ListView">
            <ListView.View>
                <GridView />
            </ListView.View>
        </ListView>
    </DockPanel>
</Window>'
$ui = Import-Xaml $xaml
```

Each column header is going to use the same event handler code; a script block is used to isolate that code. The script block begins with a `param` block, which allows the clicked header to be identified:

```
$sortHandler = {
    param ( $sender, $eventArgs )
}
```

> **Building the sort handler**
>
> The code in the following snippets is going to be placed inside the event handler; it will not be able to run independently.
>
> For example, the `$dataView` variable used in the following example will be empty.

To begin, the default sort direction for each column is set to `Ascending`:

```
$direction = 'Ascending'
```

Sorting is performed on the default view of the `ItemsSource` collection; the collection contains the results of the `Get-Process` command. The view is acquired from the `ListView` control:

```
$listView = $ui.Controls['ListView']
$dataView = [System.Windows.Data.CollectionViewSource]::GetDefaultView(
    $listView.ItemsSource
)
```

The `$dataView` object exposes a `SortDescriptions` property, which defines any sorting acting on the view of the collection; it is an array of `SortDescription`. The handler will ensure that there is only ever one `SortDescription`. The column that is currently sorted can be identified by getting the first element in the `SortDescriptions` collection:

```
$sortDescription = $dataView.SortDescriptions[0]
```

If the clicked header name matches the currently sorted property, the direction will be reversed. The direction is an enumeration that has the value `0` (`Descending`) or `1` (`Ascending`). The `-bxor` operator can be used to toggle between `0` and `1`:

```
if ($sortDescription.PropertyName -eq $sender.Content) {
    $direction = $sortDescription.Direction -bxor 1
}
```

Once the desired direction has been set all sort descriptions are removed from the view:

```
$dataView.SortDescriptions.Clear()
```

The new `SortDescription` can be added to the view, which will cause the column to sort:

```
$dataView.SortDescriptions.Add(@{
    Direction    = $direction
    PropertyName = $sender.Content
})
```

The preceding non-functional examples can now be placed inside the event handler. Each of the preceding actions runs in turn every time a column header is clicked. The complete event handler script block is shown here:

```
$sortHandler = {
    param ( $sender, $eventArgs )
    $direction = 'Ascending'
    $listView = $ui.Controls['ListView']
    $dataView = [System.Windows.Data.CollectionViewSource]::GetDefaultView(
        $listView.ItemsSource
    )
    if ($dataView.SortDescriptions) {
        $sortDescription = $dataView.SortDescriptions[0]

        if ($sortDescription.PropertyName -eq $sender.Content) {
            $direction = $sortDescription.Direction -bxor 1
        }
        $dataView.SortDescriptions.Clear()
    }
    $dataView.SortDescriptions.Add(@{
        Direction    = $direction
        PropertyName = $sender.Content
    })
}
```

The event handler is added to each of the columns as it is created and added to the `ListView`. As an event handler is being added, the header for the column must be described using a `GridViewColumnHeader` object, and the name of the header is placed in the `Content` property:

```
$ui.Controls['Button'].add_Click({
    param ( $sender, $eventArgs )
    $data = Get-Process | Select-Object Name, ID, StartTime
    $listView = $ui.Controls['ListView']
    $listView.ItemsSource = $data
    $listView.View.Columns.Clear()
    foreach ($property in $data[0].PSObject.Properties) {
        $column = @{
            DisplayMemberBinding = (
                [System.Windows.Data.Binding]$property.Name
            )
            Header               = (
                [System.Windows.Controls.GridViewColumnHeader]@{
                    Content = $property.Name
                }
            )
        }
        $listView.View.Columns.Add($column)
        $column.Header.add_Click($sortHandler)
    }
})
$ui.MainWindow.ShowDialog()
```

The outcome of the sort is shown in *Figure 16.11*:

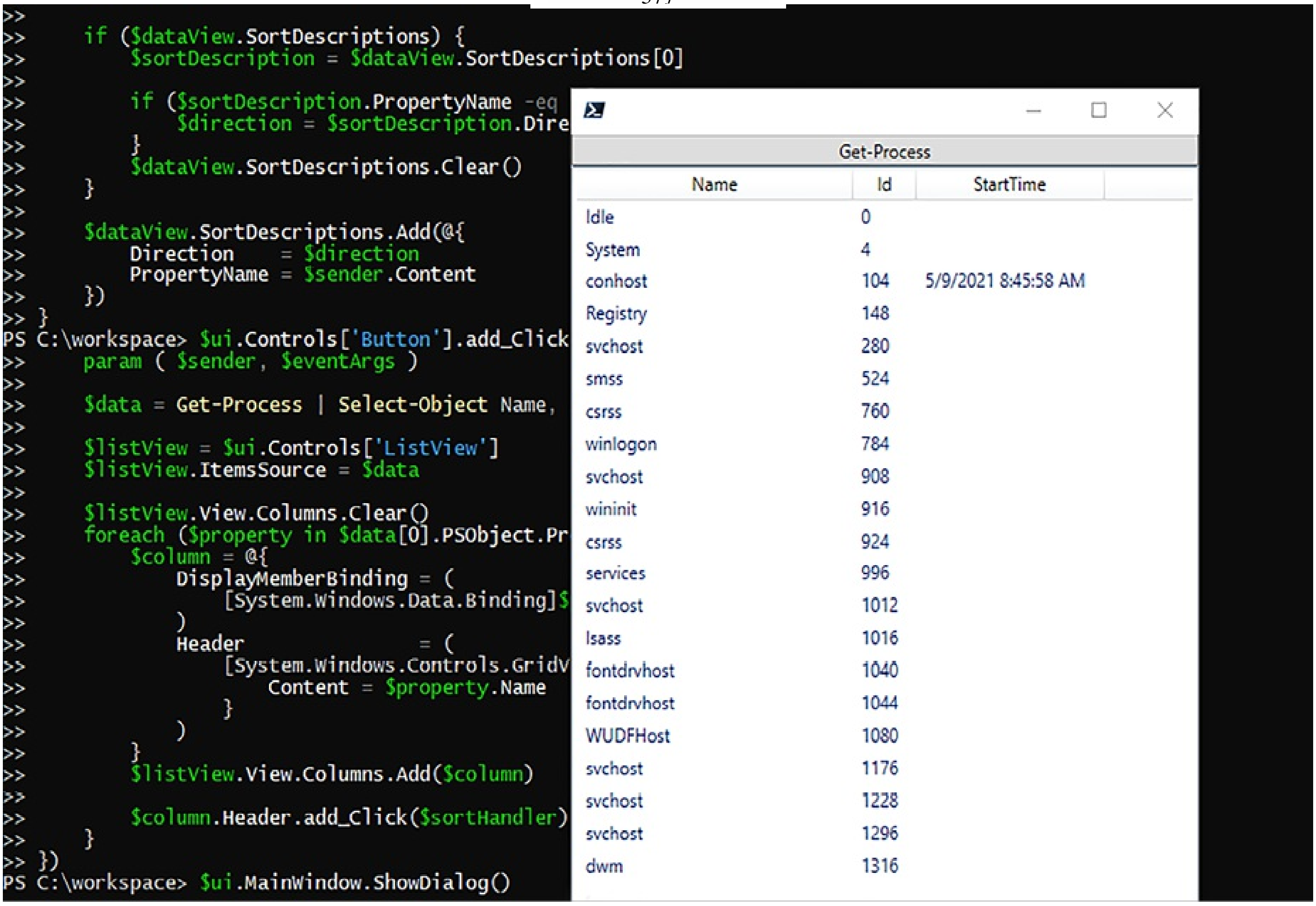


Figure 16.11: Adding column sorting to ListView

It is possible to see that the content of the list is sorted by ID by looking at the content of the list; however, normally such lists would show a visual clue in the header showing which column is sorted or which direction the property is sorted in. Adding such functionality would be a great enhancement to the UI.

One of the biggest challenges when writing user interfaces in PowerShell is maintaining a responsive interface.

Responsive interfaces

The interfaces used in all the examples so far used short-running commands that have no significant impact on the user interface.

If an event handler in the user interface carries out a longer-running activity, the entire interface will freeze. This problem is simulated by using `Start-Sleep` in the following example:

```
$xaml = '<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <Button Name="Button" Content="Start" />
</Window>'
$ui = Import-Xaml $xaml
$ui.Controls['Button'].add_Click({
    param ( $sender, $eventArgs )
    Start-Sleep -Seconds 60
})
$ui.MainWindow.ShowDialog()
```

Once the button is clicked, the user interface will freeze. It is not possible to move or close the window or interact with any of the

controls until the script that is started by clicking the button completes.

This is clearly an undesirable problem; the UI appears to have crashed and it might leave anyone using the UI wondering what has gone wrong.

Solving this problem requires the techniques used in *Chapter 15, Asynchronous Processing*, such as using runspaces, the `InitialSessionState`, and synchronized Hashtables as a means of sharing data across runspaces.

Long-running actions must be moved into the background, and the task running in the background must update the user interface running in the foreground. A PowerShell runspace will be used to carry out long-running activities.

Import-Xaml and Runspace support

The `Import-Xaml` function can be changed to better support UIs that run long-running commands.

The following changes are going to be made to the `Import-Xaml` function:

- A runspace synchronized Hashtable is created holding each of the named controls
- An `InitialSessionState` is created to share the Hashtable with the background runspace
- A PowerShell runspace is created and added as a property from the `InitialSessionState`

One final change to the function requires more of an explanation.

The background `Runspace` cannot directly interact with the UI; either to read properties from controls or to make changes. Interaction with the UI is performed via the `Dispatcher` property. The `Dispatcher` is available as a property on every single UI element and can be used to interact with elements of the UI in the UI thread.

For convenience, the `Dispatcher` property from the `Window` is added to the `Controls` Hashtable:

```
function Import-Xaml {
    param (
        [Xml]$Xaml
    )
    Add-Type -AssemblyName PresentationFramework
    $window = [System.Windows.Markup.XamlReader]::Load(
        [System.Xml.XmlNodeReader]$Xaml
    )
    $controls = [Hashtable]::Synchronized(@{
        Dispatcher = $window.Dispatcher
    })
    foreach ($control in $Xaml.SelectNodes('//*[@Name]')) {
        $controls[$control.Name] = $window.FindName($control.Name)
    }
    $initialSessionState = [InitialSessionState]::CreateDefault2()
    $initialSessionState.Variables.Add(
        [System.Management.Automation.Runspaces.SessionStateVariableEntry]::new(
            'ui',
            [PSCustomObject]@{ Controls = $controls },
            'UI controls'
        )
    )
    [PSCustomObject]@{
        MainWindow = $Window
        Controls   = $controls
        PSHost     = [PowerShell]::Create($initialSessionState)
    }
}
```

Examples and Import-Xaml

This version of `Import-Xaml` replaces the previous version. The examples that follow expect to be able to use this version of the function.

The `Dispatcher` property can be used from a runspace to read from and write to the UI.

Using the Dispatcher

As mentioned previously, the `Dispatcher` must be used in the background thread to read from the UI.

The following example writes the content of a `TextBox` to the `Information` stream using the `Write-Host` command:

```
$xaml = '<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <StackPanel>
        <TextBox Name="TextBox" Text="Hello world" />
        <Button Name="Button" Content="Start" />
    </StackPanel>
</Window>'
$ui = Import-Xaml $xaml
$ui.Controls['Button'].add_Click({
    $ui.PSHost.Commands.Clear()
    $ui.PSHost.AddScript({
        $value = $ui.Controls['Dispatcher'].Invoke(
            [Func[object]]{ $ui.Controls['TextBox'].Text }
        )
        Write-Host $value
    }).BeginInvoke()
})
$ui.MainWindow.ShowDialog()
```

In the preceding example, `AddScript` uses a script block to enclose the content to run in the background. This is cast to a string by the `AddScript` method. The `ScriptBlock` is only used to make it easier to use different quotes inside the string.

The `Click` event handler also omits the `param` block used in many of the previous examples; the `Sender` and `EventArgs` values are not used in the handler and the `param` block can be safely removed.

After the button is clicked and the UI is closed, you can use the following command to show the `Information` stream:

```
PS> $ui.PSHost.Streams.Information
Hello world
```

> **Errors in the background**
>
> Errors that occur in the background runspace will not be displayed in the PowerShell console.
>
> The `$ui.PSHost` object can be used to explore any output written by the runspace.
>
> `$ui.PSHost.Streams` includes `Error`, `Debug`, `Warning`, `Verbose`, and `Information` output.
>
> Errors that completely stop processing are written to `$ui.PSHost.InvocationStateInfo`.

The previous example makes use of the `Dispatcher`, which was added to the `Controls` Hashtable by the `Import-Xaml` function.

When requesting information from a control on the form, the following style of command may be used. This only works within the context of a running UI, and will need the body of the `Func` to be more than a comment:

```
$ui.Controls['Dispatcher'].Invoke(
    [Func[object]]{ <# Code to get UI values #> }
)
```

In the previous example, the script block is cast to the `Func[object]` type, which is used to indicate that output (of type `object`) is expected when running the script block. By default, no output is returned (even if the script block would normally have output). When setting a value in the UI, the cast to `Func[object]` can therefore be omitted if the output from the script block is not required elsewhere:

```
$ui.Controls['Dispatcher'].Invoke(
    { <# Code to set a UI value #> }
)
```

Setting a value includes any actions that make changes to the UI.

The next example updates a counter in the UI. The `Start` button is disabled while the counter is running. The UI remains relatively responsive while the loop executes; it may freeze momentarily while the `Dispatcher` executes code in the UI thread.

As before, a UI is created from a XAML document. The following UI combines some of the different layout elements used in this chapter:

```
$xaml = '<?xml version="1.0" encoding="utf-8"?>
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Width="350" Height="350">
    <DockPanel>
        <Grid DockPanel.Dock="Top">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="2*" />
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition />
                <RowDefinition />
            </Grid.RowDefinitions>
            <Label Content="Start" Margin="5,5,5,0"
             Grid.Row="0" Grid.Column="0" />
            <TextBox Name="TextBoxStart" Text="1" Margin="5,5,5,0"
             Grid.Row="0" Grid.Column="1" />
            <Label Content="End" Margin="5,5,5,0"
             Grid.Row="1" Grid.Column="0" />
            <TextBox Name="TextBoxEnd" Text="30" Margin="5,5,5,0"
             Grid.Row="1" Grid.Column="1" />
        </Grid>
        <Button Name="Button" Margin="5" Padding="5"
         Content="Go" DockPanel.Dock="Bottom" />
        <Label Name="Label"
         Margin="5" HorizontalContentAlignment="Center"
         VerticalContentAlignment="Center" FontSize="32"
        />
    </DockPanel>
</Window>'
$ui = Import-Xaml $xaml
```

The event handler for the `Click` event on the button describes how the form is changed:

```
$ui.Controls['Button'].add_Click({
    $ui.PSHost.Commands.Clear()
    $ui.PSHost.AddScript({
        $dispatcher = $ui.Controls['Dispatcher']
        $dispatcher.Invoke( {
            $ui.Controls['Button'].IsEnabled = $false
        })
        $start = $dispatcher.Invoke([Func[object]]{
            $ui.Controls['TextBoxStart'].Text -as [int]
        })
        $end = $dispatcher.Invoke([Func[object]]{
            $ui.Controls['TextBoxEnd'].Text -as [int]
        })
        foreach ($number in $start..$end) {
            $ui.Controls['Dispatcher'].Invoke({
                $ui.Controls['Label'].Content = $number
            })
            Start-Sleep -Seconds 1
        }
        $ui.Controls['Dispatcher'].Invoke({
            $ui.Controls['Button'].IsEnabled = $true
        })
    }).BeginInvoke()
})
$ui.MainWindow.ShowDialog()
```

Implemented in this way, the UI will remain generally responsive. The preceding UI is shown in *Figure 16.12* after the **Go** button has been clicked:

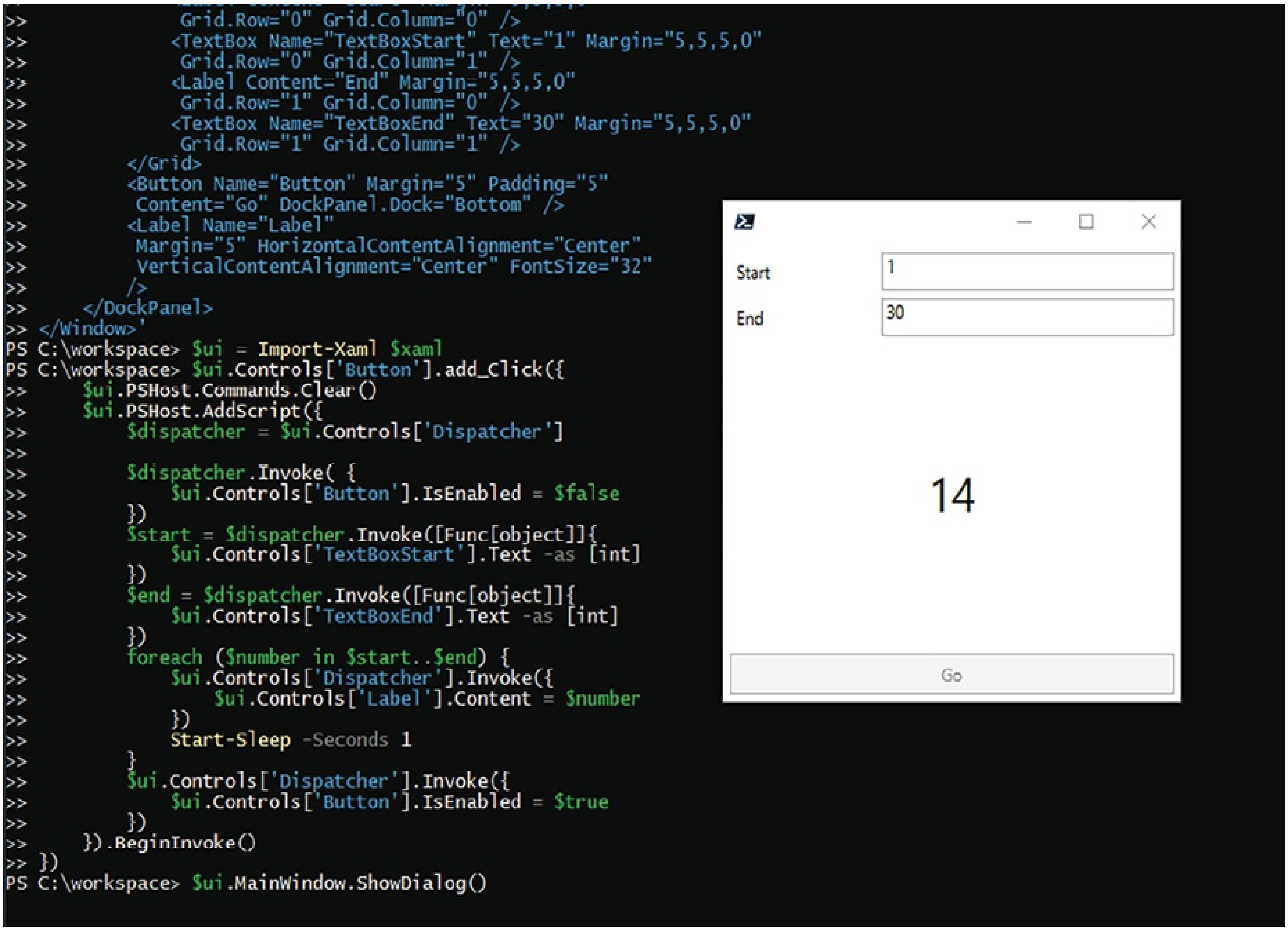


Figure 16.12: Counter UI

It is important to keep long-running activities outside of the code running in the `Dispatcher`.

The `Dispatcher` can be used on either side of the process to enable and disable controls, to write status messages and output, and so on.

Summary

User interfaces are a common requirement in PowerShell, and even though PowerShell is written to run as a shell it is still possible to create advanced user interfaces.

WPF includes a wide variety of different controls that can be combined to build a user interface. The initial layout of a WPF UI can be described in a XAML document, reducing the amount of code required to create the interface. A designer can be used to help generate the XAML content if required.

Simple controls such as `Grid`, `StackPanel`, and `DockPanel` can be used to place controls without a need to resort to absolute or coordinate-based positioning within an interface.

Controls can be given names within the XAML document allowing PowerShell to find controls to attach event handlers or change values at runtime.

Elements of the user interface can be added or changed in event handlers. A `ListView` control was used to demonstrate a dynamically created view of an object created in PowerShell.

Making a responsive user interface is challenging in PowerShell. The `Dispatcher` in WPF can be used to interact with the UI thread from a `Runspace` running in the background. The `Dispatcher` is used to read from and write to the user interface.

The next chapter explores scripts, functions, and script blocks.

17

Scripts, Functions, and Script Blocks

In PowerShell, scripts and functions are commands. They have overlapping capabilities and often only differ by common use.

Scripts tend to be used to implement well-defined processes; such scripts often run on a schedule of some kind. A script may include functions or run other scripts, and a script will often use commands from other modules. A script is convenient as it consists of a single `.ps1` file.

You can also use a function to implement a single process in the same way as a script. The function might be part of a script, implementing a process, or part of a module. When automating a process, writing a module to host a function might be regarded as slightly less convenient and less simple than writing a script.

Whether a function is in a script or a module, the function should strive to be simple. It should aim to be great at one job, and, if possible, reusable. Functions are the building blocks of more complex code, including scripts and other functions.

A script block also has overlapping capabilities with scripts and functions. A script block is an anonymous or unnamed function; this is also known as **lambda**.

Filters are briefly discussed in this chapter while exploring the named blocks, `begin`, `process`, and `end`. A filter is a specialized function, a throwback to PowerShell 1. Filters are rarely used and are not recommended for general use because of their rarity.

This chapter explores the following topics:

- About style
- Capabilities of scripts, functions, and script blocks
- Parameters and the `param` block
- The `CmdletBinding` attribute
- The `Alias` attribute
- `begin`, `process`, `end`, and `cleanup`
- Managing output
- Working with long lines
- Comment-based help

Style in PowerShell is a very subjective topic, but an important one when writing code that will be maintained over a long period of time.

About style

Style in the context of a scripting language is all about aesthetics; it describes how variables are named, how code is indented, where opening and closing brackets are placed, and so on. Style is an easy thing to disagree with; the topic is full of opinions, many of which come down to what an individual feels is right.

PowerShell does not have an official style, nor does it have an official best practice.

PowerShell does have a set of guidelines for cmdlet development (commands typically written in a compiled language like C#): https://docs.microsoft.com/powershell/scripting/developer/cmdlet/cmdlet-development-guidelines?view=powershell-7.

It has been left to the community of PowerShell users and developers to provide further guidance. The community-created PowerShell Practice and Style repository describes many of the most widely accepted conventions: https://github.com/poshcode/powershellpracticeandstyle.

Many of these conventions, for example, warnings about the use of aliases in production code, are implemented in the `PSScriptAnalyzer` module. The `PSScriptAnalyzer` module is integrated with the PowerShell extension in Visual Studio Code and will show warnings as code is written:

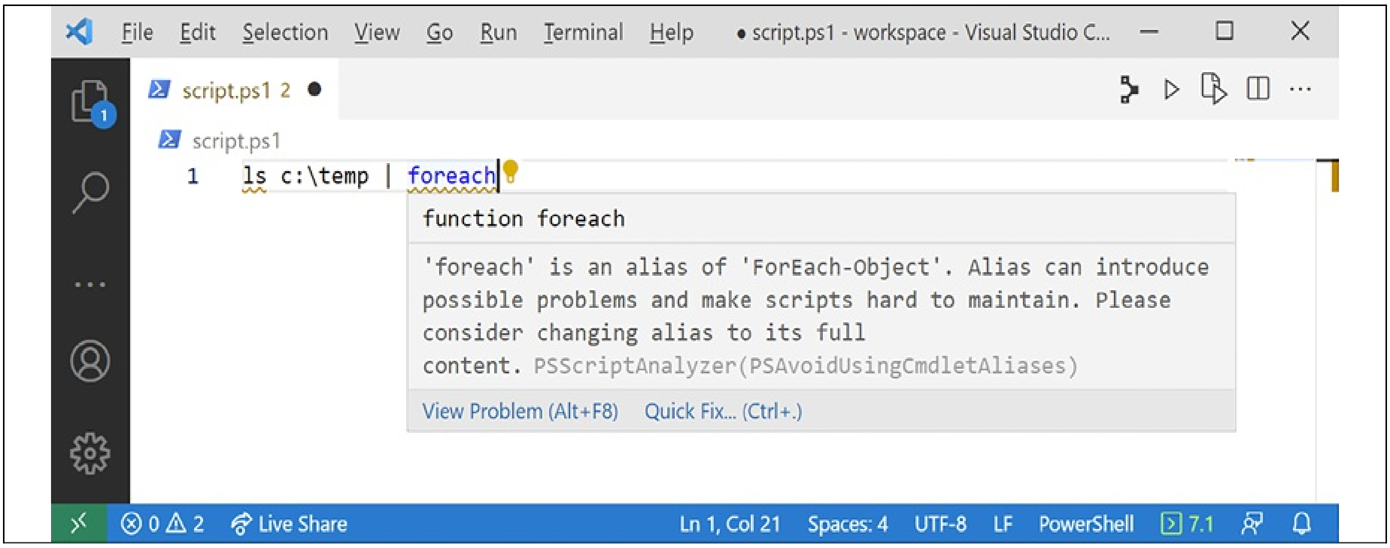


Figure 17.1: Warning from PSScriptAnalyzer in Visual Studio Code

A warning from `PSScriptAnalyzer` in Visual Studio Code often provides a link to fix the perceived problem. Not everyone will agree with all the rules and this is fine. The `PSScriptAnalyzer` repository includes instructions for suppressing rules: https://github.com/PowerShell/PSScriptAnalyzer/blob/master/README.md.

The guidance on styling is therefore simple:

- Choose a style and use that style consistently. Remember that code will likely need to be maintained in the future.
- If contributing to a project started by somebody else, be considerate of any existing style; use that style in any changes.

Personal style may change over time. This too is normal.

Establishing a style when writing code as a team can be difficult. Style choices can be a very personal topic. Compromise may be needed to find an approach that works. Having established something of a style, scripts and functions can be written, starting with an exploration of the capabilities of scripts and functions. Ideally, the style chosen should be documented and enforced using the styling rules available with `PSScriptAnalyzer`.

Capabilities of scripts, functions, and script blocks

Scripts, functions, and script blocks share many of the same capabilities. These capabilities are explored during this and the next few chapters. Scripts, functions, and script blocks can each:

- Define parameters
- Support pipeline input
- Support common parameters, including support for `Confirm` and `WhatIf`
- Allow other functions to be nested inside

Scripts, but not functions or script blocks, support the `using assembly` and `using namespace` directives. Scripts also support the `#Requires` statement.

Scripts and using statements

`using` statements, introduced with PowerShell 5, were described in *Chapter 7*, *Working with .NET*. A short example of a `using` statement is shown here:

```
using namespace System.Xml.Linq
```

A function may benefit from `using` statements declared in a parent scope. The parent scope includes code run on the console, a script that contains a function, or a module (`.psm1` file) that contains a function.

The order in which the statements appear is not important, meaning the order of the commands in the previous example can be reversed with no change in the result. `using namespace` is a passive statement and is only important when code utilizing a shortened namespace is run.

Scripts and Requires

The `Requires` statement is valid only in scripts and can be used to restrict a script from running if certain conditions are not met. For example, a script may require administrative rights or certain modules.

The `Requires` statement must be the first line in the script; comments and other code may not appear before it. You can use more than one `Requires` statement may if required.

An example of a `Requires` statement is shown here:

```
#Requires -RunAsAdministrator
#Requires -Modules @{ ModuleName = 'TLS'; ModuleVersion = '2.0.0' }
```

Notice that there is no space between the comment character, `#`, and the `Requires` keyword.

PowerShell includes help for the `Requires` statement that describes the possible parameters:

```
Get-Help about_Requires
```

In a script, the `Requires` statement may be used to declare a need for administrative rights, a certain version of PowerShell, or the presence of certain modules.

Nesting functions

In the same way that a script can contain functions, a function can contain other functions. The following function contains two other functions; those functions are only available within the scope of `function Outer`:

```
function Outer {
    param (
        $Parameter1
    )
    function Inner1 {
    }
    function Inner2 {
    }
    Write-Host 'Hello world'
}
```

Nested functions allow an author to isolate repeated sections of code with a function.

Nested functions must appear before they are used, but otherwise can appear anywhere in the body of the function.

The disadvantage of nesting a function is that it becomes harder to debug. The function only exists in the context of its parent function; it cannot be called and tested from the scope above that. This is an important consideration when developing a function as part of a larger work and it is therefore advisable to avoid nesting one function within another.

Script blocks and closures

A script block is an anonymous function. Script blocks are used a great deal in PowerShell; for example, they are used with:

- `Where-Object` as `FilterScript`

- `ForEach-Object` for the `begin`, `process`, and `end` parameters
- `Select-Object` as an `Expression` for a calculated property

Like a function, when a script block is run, it can use variables from a parent scope. The following example will return the string `"first value"`, the value of the `$string` variable at the point in time it is run:

```
$string = "first value"
$scriptBlock = { $string }
& $scriptBlock
```

If the string changes before `scriptBlock` is run, the output of `scriptBlock` will change, in this case to `"second value"`:

```
$scriptBlock = { $string }
$string = "second value"
& $scriptBlock
```

The `GetNewClosure` method can be used to copy the values of variables from the current session into the `scriptBlock` session. The following example will therefore return the value `'first value'`, despite the variable being updated afterward:

```
$string = 'first value'
$scriptBlock = { $string }.GetNewClosure()
$string = 'second value'
& $scriptBlock
```

Closures are an interesting facet of script blocks in PowerShell, but it is difficult to find an instructive use that can be easily demonstrated without a great deal of design context.

The capabilities listed in this section are less frequently used than the following major features, such as our next topic on parameters.

Parameters and the param block

You can use parameters to describe and give names to the values a command is willing to accept when it is run. The list of parameters is separated by a comma.

Parameters can be defined as a block using the `param` keyword, which is the most popular approach as parameter blocks in PowerShell can become quite large. Using the `param` keyword is the only way to describe parameters for scripts and script blocks:

```
param (
    $Parameter1,
    $Parameter2
)
```

When used in a function, the `param` block is used as follows:

```
function New-Function {
    param (
        $parameter1,
        $parameter2
    )
}
```

The `param` block is required if `CmdletBinding`, `Alias`, or any other attributes are applied to the function. The `CmdletBinding` and `Alias` attributes are explored later in this chapter.

Functions also allow parameters to be defined immediately after the function name. For example:

```
function New-Function($Parameter1, $Parameter2) {
    # Function body
}
```

When used, the `param` block must appear before all other code. An exception to this is `using` statements in a script, which must be written before `param`.

By default, parameters have the `System.Object` type. This means that any value type can be passed into a parameter. It may be desirable to restrict values to those of a specific type.

Parameter types

The type assigned to a parameter is written before the parameter name. A line break may appear after the type if desired.

For example, if the function expects a string, the parameter type might be set to `[string]`:

```
param (
    [string]$Parameter1
)
```

Any value passed as an argument will be converted into a string regardless of the original type.

When assigning a type to a parameter, the type persists until a new type is assigned or the variable is removed (for example, with `Remove-Variable`). Any values assigned to `$Parameter1` from the preceding example will be converted to a string.

Parameters that are given a type will be initialized with a default value or may be given an explicit default value.

Default values

Default values are used when a parameter value is not explicitly passed when calling the script, function, or script block.

Value types, such as `Boolean` or `Int` (or `Int32`), and other numeric types are initialized with a default value for that type. A parameter of the `Boolean` type can never be `null`; it will default to `false`. Numeric values will default to `0`, a `String` parameter will default to an empty string, and so on.

Conversely, parameters using types such as `DateTime`, `TimeSpan`, and `Hashtable` will default to `null`.

You can give parameters explicit default values by assigning a value in the `param` block; for example:

```
function Test-Parameter {
    param (
        [string]$Parameter1 = 'DefaultValue'
    )
}
```

If the assignment is the result of a command, the command must be placed in parentheses:

```
function Test-Parameter {
    param (
        [string]$ProcessName = (Get-Process -Id $PID |
            Select-Object -ExpandProperty Name)
    )
}
```

Assigning a default value was the basis for making parameters mandatory in PowerShell 1. The parameter would be assigned a `throw` statement by default. This is rarely seen in modern code; for example:

```
function Test-Parameter {
    param (
        [string]$Parameter1 = $(throw 'This parameter is mandatory')
    )
}
```

This method of making parameters mandatory was replaced in PowerShell 2 by the `Mandatory` property of the `Parameter` attribute. The `Parameter` attribute is used to define behavior for a parameter and is explored in detail in *Chapter 18, Parameters, Validation, and Dynamic Parameters*.

It is possible to provide a default value for a parameter based on the value of another within the `param` block.

Cross-referencing parameters

When using a `param` block, it is possible to cross-reference parameters. That is, the default value of a parameter can be based on the value of another parameter as shown in the following example:

```
function Get-Substring {
    param (
        [string]$String,
        [int]$Start,
        [int]$Length = ($String.Length - $Start)
    )
    $String.Substring($Start, $Length)
}
```

The value of the `Length` parameter will use the default, derived from the first two parameters, unless the user of the function supplies their own value. The order of the parameters is important: the `Start` parameter must be declared in the `param` block before it can be used in the default value for `Length`.

Parameters in PowerShell are complex, and an exploration of their capabilities is covered in *Chapter 18, Parameters, Validation, and Dynamic Parameters*. This chapter focuses on the structure of strings and functions starting at the top with `CmdletBinding`.

The CmdletBinding attribute

The `CmdletBinding` attribute is used to turn a function into an advanced function and is placed immediately above a `param` block. The attribute is used to add extra functionality, such as access to common parameters, control over the impact level, and so on. Scripts are not referred to as advanced scripts, but the same principle applies.

`CmdletBinding` may be used to do the following:

- Add common parameters, such as `ErrorAction`, `Verbose`, `Debug`, `ErrorVariable`, `WarningVariable`, and so on
- Enable use of the built-in `$PSCmdlet` variable
- Declare support for `WhatIf` and `Confirm` and define the impact level of the command

If a script or function has no parameters, and wishes to make use of the capabilities provided by `CmdletBinding`, an empty `param` block must be declared:

```
function Test-EmptyParam {
    [CmdletBinding()]
    param ( )
}
```

One of the most important features added by `CmdletBinding` are the common parameters.

Common parameters

With `CmdletBinding` in place, a script or function may use common parameters. The common parameters are listed here:

- `Debug`
- `ErrorAction`
- `ErrorVariable`
- `InformationAction`
- `InformationVariable`
- `OutVariable`
- `OutBuffer`
- `PipelineVariable`
- `Verbose`
- `WarningAction`
- `WarningVariable`

PowerShell provides a description of each of the common parameters in the `about_CommonParameters` help file:

```
Get-Help about_CommonParameters
```

For example, the `Verbose` parameter is made available. The following function will display the verbose message when it is run with the `Verbose` parameter:

```
function Show-Verbose {
    [CmdletBinding()]
    param ( )
    Write-Verbose 'Verbose message'
}
```

In a similar way, parameters such as `ErrorAction` will affect `Write-Error` if it is used within the function.

`CmdletBinding` itself provides several properties used to influence the behavior of a function or script.

CmdletBinding properties

The full set of possible values that may be assigned can be viewed by creating an instance of the `CmdletBinding` object:

```
PS> [CmdletBinding]::new()
PositionalBinding       : True
DefaultParameterSetName :
SupportsShouldProcess   : False
SupportsPaging          : False
SupportsTransactions    : False
ConfirmImpact           : Medium
HelpUri                 :
RemotingCapability      : PowerShell
TypeId                  : System.Management.Automation.CmdletBindingAttribute
```

For example, the output from the preceding command shows the existence of a `PositionalBinding` property. Setting this to `false` disables automatic position binding:

```
function Test-Binding {
    [CmdletBinding(PositionalBinding = $false)]
    param (
        $Parameter1
    )
}
```

When the preceding function is called, and a value for `Parameter1` is given by position, an error is thrown:

```
PS> Test-Binding 'value'
Test-Binding : A positional parameter cannot be found that accepts argument 'value'.
```

The most used properties of `CmdletBinding` are `SupportsShouldProcess` and `DefaultParameterSetName`. `DefaultParameterSetName` are explored in *Chapter 18, Parameters, Validation, and Dynamic Parameters*. `ShouldProcess` and `ShouldContinue` offer the ability to raise a prompt for a user to confirm an action.

ShouldProcess and ShouldContinue

Setting the `SupportsShouldProcess` property of `CmdletBinding` enables the `ShouldProcess` parameters, `Confirm` and `WhatIf`. These two parameters are used in conjunction with the `ShouldProcess` method, which is exposed on the `$PSCmdlet` variable, an automatically created variable available in the body of an advanced function or script.

Both the `ShouldProcess` and `ShouldContinue` methods of the `$PSCmdlet` variable become available when a script or function has the `CmdletBinding` attribute and the `SupportsShouldProcess` property is set. The following example enables the `SupportsShouldProcess` setting:

```
function Enable-ShouldProcess {
    [CmdletBinding(SupportsShouldProcess)]
    param ( )
}
```

The `ShouldProcess` method is more commonly used than `ShouldContinue`.

ShouldProcess

`ShouldProcess` is used to support `WhatIf` and is responsible for raising confirmation prompts based on the impact level of a command.

The following example will display a message instead of running the `Write-Host` statement when the `WhatIf` parameter is used:

```
function Test-ShouldProcess {
    [CmdletBinding(SupportsShouldProcess)]
    param ( )
    if ($pscmdlet.ShouldProcess('SomeObject')) {
        Write-Host 'Deleting SomeObject' -ForegroundColor Cyan
    }
}
```

When run using the `WhatIf` parameter, the command will show the following message:

```
PS> Test-ShouldProcess -WhatIf
What if: Performing the operation "Test-ShouldProcess" on target "SomeObject".
```

The name of the operation, which defaults to the command name, can be changed using a second overload for `ShouldProcess`:

```
function Test-ShouldProcess {
    [CmdletBinding(SupportsShouldProcess)]
    param ( )
    if ($pscmdlet.ShouldProcess('SomeObject', 'delete')) {
        Write-Host 'Deleting SomeObject' -ForegroundColor Cyan
    }
}
```

This would change the message to the following:

```
PS> Test-ShouldProcess -WhatIf
What if: Performing the operation "delete" on target "SomeObject".
```

The whole message can be changed by adding one extra argument:

```
function Test-ShouldProcess {
    [CmdletBinding(SupportsShouldProcess)]
    param ( )
    if ($pscmdlet.ShouldProcess(
            'Message displayed using WhatIf',
            'Warning: Deleting SomeObject',
            'Question: Are you sure you want to do continue?')) {
        Write-Host 'Deleting SomeObject' -ForegroundColor Cyan
    }
}
```

Using the `Confirm` parameter instead of `WhatIf` forces the appearance of the second two messages and adds a prompt:

```
PS> Test-ShouldProcess -Confirm
Question: Are you sure you want to do continue?
Warning: Deleting SomeObject
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"):
```

The different responses are automatically available without further code. If the request is within a loop, `Yes to All` may be used to bypass additional prompts. Replying `Yes to All` applies to all instances of `ShouldProcess` in the script or function:

```
function Test-ShouldProcess {
    [CmdletBinding(SupportsShouldProcess)]
    param ( )
    foreach ($value in 1..2) {
        if ($pscmdlet.ShouldProcess(
                "Would delete SomeObject $value",
                "Warning: Deleting SomeObject $value",
                'Question: Are you sure you want to do continue?')) {
            Write-Host "Deleting SomeObject $value" -ForegroundColor Cyan
        }
    }
}
```

Whether or not the confirmation prompt is displayed depends on the comparison between `ConfirmImpact` (`Medium` by default) and the value in the `$ConfirmPreference` variable, which is `High` by default.

If the impact of the function is raised to `High`, the prompt will display by default instead of on-demand. This is achieved by modifying the `ConfirmImpact` property of the `CmdletBinding` attribute:

```
function Test-ShouldProcess {
    [CmdletBinding(SupportsShouldProcess, ConfirmImpact = 'High')]
    param ( )
    if ($pscmdlet.ShouldProcess('SomeObject')) {
        Write-Host 'Deleting SomeObject' -ForegroundColor Cyan
    }
}
```

When the function is executed, the confirmation prompt will appear unless the user either uses `-Confirm:$false` or sets `$ConfirmPreference` to `None`.

`ShouldContinue` is available as an alternative to `ShouldProcess`.

ShouldContinue

The `ShouldContinue` method is also made available when the `SupportsShouldProcess` property is set in `CmdletBinding`.

`ShouldContinue` differs from `ShouldProcess` in that it always prompts. This technique is used by commands such as `Remove-Item` to force a prompt when attempting to delete a directory without supplying the `Recurse` parameter.

As the prompt for `ShouldContinue` is forced, it is rarely a better choice than `ShouldProcess`. It is available for cases where a function must have a confirmation prompt that cannot be bypassed. Using `ShouldContinue` may make it impossible to run a function without user interaction unless another means of bypassing the prompt is created.

The use of the `ShouldContinue` method is like the `ShouldProcess` method. The most significant difference is that the `Yes to All` and `No to All` options are not automatically implemented. The following example does not offer `Yes to All` and `No to All` options:

```
function Test-ShouldContinue {
    [CmdletBinding(SupportsShouldProcess)]
    param ( )
    if ($pscmdlet.ShouldContinue(
            "Warning: Deleting SomeObject $value",
            'Question: Are you sure you want to do continue?')) {
        Write-Host 'Deleting SomeObject' -ForegroundColor Cyan
    }
}
```

The prompt displayed is shown as follows:

```
PS> Test-ShouldContinue
Question: Are you sure you want to do continue?
Warning: Deleting SomeObject
[Y] Yes [N] No [S] Suspend [?] Help (default is "Y"): y
```

Adding support for `Yes to All` and `No to All` means using three extra arguments for the `ShouldContinue` method. The first of these new arguments, `hasSecurityImpact`, affects whether the default option presented is `Yes` (when `hasSecurityImpact` is `false`) or `No` (when `hasSecurityImpact` is `true`).

Boolean variables must be created to hold the `Yes to All` and `No to All` responses. These are passed as reference arguments to the `ShouldContinue` method as shown in the following example:

```
function Test-ShouldContinue {
    [CmdletBinding(SupportsShouldProcess)]
    param ( )
    $yesToAll = $noToAll = $false
    if ($pscmdlet.ShouldContinue(
            "Warning: Deleting SomeObject $value",
            'Question: Are you sure you want to do continue?',
            $false,
            [Ref]$yesToAll,
            [Ref]$noToAll)) {
        Write-Host 'Deleting SomeObject' -ForegroundColor Cyan
    }
}
```

The confirmation prompt will now include the `Yes to All` and `No to All` options:

```
PS> Test-ShouldContinue
Question: Are you sure you want to do continue?
Warning: Deleting SomeObject
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"):
```

It is possible to provide a means of bypassing the prompt by implementing another switch parameter. For example, a `Force` parameter might be added:

```
function Test-ShouldContinue {
    [CmdletBinding(SupportsShouldProcess)]
    param (
        [Switch]$Force
    )
    $yesToAll = $noToAll = $false
    if ($Force -or $pscmdlet.ShouldContinue(
            "Warning: Deleting SomeObject $value",
            'Question: Are you sure you want to do continue?',
            $false,
            [Ref]$yesToAll,
            [Ref]$noToAll)) {
        Write-Host 'Deleting SomeObject' -ForegroundColor Cyan
    }
}
```

As the value of `Force` is evaluated before `ShouldContinue`, the `ShouldContinue` method will not run if the `Force` parameter is supplied.

You can add other attributes to the preceding the `param` block in addition to `CmdletBinding`, such as `Alias`.

The Alias attribute

Like `CmdletBinding`, the `Alias` attribute may be placed preceding the `param` block. It may be used before or after `CmdletBinding`; the ordering of attributes does not matter.

The `Alias` attribute is used to optionally create aliases for a function. A function can have one or more aliases; the attribute accepts either a single value or an array of values. For example, you can create an alias `gsm` for the function `Get-Something` via the following attribute:

```
function Get-Something {
    [CmdletBinding()]
    [Alias('gsm')]
    param ( )
}
```

The body of a function may use named blocks, such as `begin`, `process`, and `end`, to control how code executes.

begin, process, end, and cleanup

A script or function often begins with comment-based help, followed by a `param` block. Following this, one or more named blocks may be used.

The named blocks are:

- `begin`
- `process`
- `end`
- `dynamicparam`
- `cleanup`

The `dynamicparam` block is explored in *Chapter 18, Parameters, Validation, and Dynamic Parameters*, as it is more complex and ties to more advanced parameter usage than covered by this chapter.

The `cleanup` block is also briefly introduced although this has still not found its way into the preview versions of PowerShell 7 at the time of writing.

In a script or function, if none of these blocks are declared, content is in the `end` block.

The named blocks refer to a point in a pipeline and therefore make the most sense if the command is working on pipeline input.

In a filter, if none of these blocks are declared, content is in the `process` block. This is the only difference between a function and a filter.

This difference in a default block is shown in the following pipeline example. The function must explicitly declare a `process` block to use the `$_` variable. The filter can leverage the default block:

```
PS> function first { process { $_ } } # end block by default
PS> filter second { $_ }              # process block by default
PS> 1..2 | first                      # explicit process
1
2
PS> 1..2 | second                     # implicit process
1
2
```

Using named blocks is optional. If a command is not expected to work on a pipeline, the content can be left to fall into the default block.

begin

The `begin` block runs before pipeline processing starts. A pipeline that contains several commands will run each of the `begin` blocks for each command in turn before taking any further action.

The following example shows a short function with a `begin` block:

```
function Show-Pipeline {
    begin {
        Write-Host 'Pipeline start'
    }
}
```

Pipeline input, or values from parameters accepting pipeline input, is not available within the `begin` block as pipeline input has not been passed into the function yet.

`begin` can be used to create things that are reused by the process block, setting up the initial conditions for a loop perhaps.

process

The `process` block runs once for each value received from the pipeline. The built-in variable `$_` or `$PSItem` may be used to access objects in the pipeline within the `process` block. In the following example, the automatic variable `$MyInvocation` is used to show the position in a pipeline of the command:

```
function Show-Pipeline {
    begin {
        $position = $MyInvocation.PipelinePosition
        Write-Host "Pipeline position ${position}: Start"
    }
    process {
        Write-Host "Pipeline position ${position}: $_"
        $_
    }
}
```

When an object is passed to the pipeline, the start message will be shown before the numeric value:

```
PS> $result = 1..2 | Show-Pipeline
Pipeline position 1: Start
Pipeline position 1: 1
Pipeline position 1: 2
```

Adding `Show-Pipeline` to the end of the pipeline will show that `begin` executes twice before `process` runs:

```
PS> $result = 1..2 | Show-Pipeline | Show-Pipeline
Pipeline position 1: Start
Pipeline position 2: Start
Pipeline position 1: 1
Pipeline position 2: 1
Pipeline position 1: 2
Pipeline position 2: 2
```

The `$result` variable will contain the output of the last `Show-Pipeline` command.

end

The `end` block executes after `process` has acted on all objects in the input pipeline.

In the `end` block, `$_` will be set to the last value received from a pipeline. Any other parameters that accept pipeline input will be filled with the last value from the `process` block:

```
function Show-Pipeline {
    begin {
        $position = $myinvocation.PipelinePosition
        Write-Host "Pipeline position ${position}: Start"
    }
    process {
        Write-Host "Pipeline position ${position}: $_"
        $_
    }
    end {
        Write-Host "Pipeline position ${position}: End"
    }
}
```

Running this command in a pipeline shows `end` executing after all items in the input pipeline have been processed:

```
PS> $result = 1..2 | Show-Pipeline
Pipeline position 1: Start
Pipeline position 1: 1
Pipeline position 1: 2
Pipeline position 1: End
```

Commands that make extensive use of the `end` block include `Measure-Object`, `ConvertTo-Html`, and `ConvertTo-Json`. Such commands cannot return output until the end because the output is only valid when complete. The same is true of any other command that must gather input during a `process` block, and output something on completion.

A simple command to count the number of elements in an input pipeline is shown here. The `process` block is unable to determine this; it must run again and again until the input pipeline is exhausted:

```
function Measure-Item {
    begin {
        $count = 0
    }
    process {
        $count++
    }
    end {
        $count
    }
}
```

The `end` block will run after all values in a pipeline have been received unless a terminating error is thrown by an earlier block.

cleanup

The `cleanup` block is coming to PowerShell, although it is not quite clear when.

> **Not in PowerShell 7 yet? Then why here?**
>
> The `cleanup` block will be an incredibly useful feature once it finally finds a way into PowerShell.
>
> https://github.com/PowerShell/PowerShell/pull/9900
>
> The pull request, a request to make a change to PowerShell itself, has been open for over a year at the time of writing. The author of the pull request is actively responding to issues raised by the PowerShell team.
>
> This section is here first and foremost because I expected it to be merged by now.
>
> I still have every confidence it will be merged and find a way into a future preview release. It is, unfortunately, just not quite clear when that might happen.

The `cleanup` block addresses a long-standing problem when using pipelines with persistent streams or connections in PowerShell.

The following example is implemented in such a way that the `Remove-Item` command will occasionally fail:

```
function Invoke-Something {
    begin {
         $fileStream = [System.IO.File]::OpenWrite((
            Join-Path -Path $pwd -ChildPath NewFile.txt
        ))
        $count = 0
    }
    process {
        if ((++$count) -eq 3) {
            throw 'Oh no! Something unexpected went wrong'
        }
    }
    end {
        $fileStream.Close()
    }
}
1..5 | Invoke-Something
Remove-Item NewFile.txt
```

`Remove-Item` fails because the `$fileStream` stream is not properly closed by the function unless the `end` block executes. The `end` block will not execute if a terminating error is raised inside the `process` block, prematurely stopping the command. .NET garbage collection must run to destroy the unreferenced file stream object before `Remove-Item` can act.

The `cleanup` block addresses this problem as it will always run, even if a terminating error is raised during the normal

operation of the command:

```
function Invoke-Something {
    begin {
         $fileStream = [System.IO.File]::OpenWrite((
            Join-Path -Path $pwd -ChildPath NewFile.txt
        ))
        $count = 0
    }
    process {
        if ((++$count) -eq 3) {
            throw 'Oh no! Something unexpected went wrong'
        }
    }
    end {
        # Any actions end needs to perform
    }
    cleanup {
        $fileStream.Close()
    }
}
```

The `cleanup` block can therefore be used to properly close any open streams or connections, even if the command itself fails for some reason.

`begin`, `process`, and `end` can each emit values to the output pipeline. `dynamicparam` and `cleanup` cannot send values to an output pipeline. The `return` keyword is sometimes mistaken as being capable of limiting the output of a command.

Named blocks and return

You can use the `return` keyword to gracefully end the execution of a piece of a script block.

The `return` keyword is often confused with `return` in C#, where it explicitly returns an object from a method. In PowerShell, `return` has a slightly different purpose.

When a named block is executing, the `return` keyword may be used to end the execution of a script block immediately without stopping the rest of the pipeline.

For example, a `return` statement in the `process` block might be used to end early in certain cases. The `end` block will continue to execute as normal:

```
function Invoke-Return {
    process {
        if ($_ -gt 2 -and $_ -lt 9) {
            return
        }
        $_
    }
    end {
        'All done'
    }
}
```

When run, the `process` block will end early when the condition is met:

```
PS> 1..10 | Invoke-Return
1
2
9
10
All done
```

The `return` keyword may have a value to return; however, there is no effective difference between the two examples here:

```
function Test-Return { return 1 }
function Test-Return { 1; return }
```

In each case, `1` is sent to the output pipeline and then the function ends. `return` does not constrain or dictate the output from a function or a script.

`begin`, `process`, and `end` may all send output that has to be managed.

Managing output

PowerShell does not have a means of strictly enforcing the output from a script or function.

Any statement, composed of any number of commands, variables, properties, and method calls, may generate output. This output will be automatically sent to the output pipeline by PowerShell as it is generated. Unanticipated output can cause bugs in code.

The following function makes use of the `StringBuilder` type. Many of the methods in `StringBuilder` return the `StringBuilder` instance. This is shown here:

```
PS> using namespace System.Text
PS> $stringBuilder = [StringBuilder]::new()
PS> $stringBuilder.AppendLine('First')
Capacity    MaxCapacity    Length
--------    -----------    ------
      16     2147483647         7
```

This is useful in that it allows chaining to build up a more complex string in a single statement. The following function makes use of that chaining to build up a string:

```
using namespace System.Text
function Get-FirstService {
    $service = Get-Service | Select-Object -First 1
    $stringBuilder = [StringBuilder]::new()
    $stringBuilder.AppendFormat('Name: {0}', $service.Name).
        AppendLine().
        AppendFormat('Status: {0}', $service.Status).
        AppendLine().
        AppendLine()
    $stringBuilder.ToString()
}
```

When the function is run, both the `StringBuilder` object and the assembled string will be written to the output pipeline:

```
PS> Get-FirstService
Capacity    MaxCapacity    Length
--------    -----------    ------
      64     2147483647        37
Name: aciseagent
Status: Running
```

This example is contrived and writing the function slightly differently would resolve the problem of it emitting unwanted output. However, this problem is not unique to the type used here.

When writing a function or script, it is important to be aware of the output of the statements used. If a statement generates output, and that output is not needed, it must be explicitly discarded. PowerShell will not automatically discard output from commands in functions and scripts.

There are several techniques available for dropping unwanted output; the following subsections show the common approaches.

The Out-Null command

The `Out-Null` command can be used at the end of a pipeline to discard the output from the preceding pipeline.

The `Out-Null` command is relatively unpopular in Windows PowerShell as it is slow. As a command in Windows PowerShell, it must still act on input passed to it, which has a processing cost.

In PowerShell 6 and 7, the speed issue is resolved; `Out-Null` is no longer quite what it seems. The `Out-Null` command acts as a keyword for the parser. When the parser encounters the keyword, any output piped to the command is discarded without invoking `Out-Null` as a command. This change in implementation makes `Out-Null` one of the fastest, if not the fastest, of the available options for discarding unwanted output.

Sticking with the `StringBuilder` example, the unwanted value might have dropped by appending `Out-Null`, as the following shows:

```
using namespace System.Text
$service = Get-Service | Select-Object -First 1
$stringBuilder = [StringBuilder]::new()
$stringBuilder.
    AppendFormat('Name: {0}', $service.Name).AppendLine().
    AppendFormat('Status: {0}', $service.Status).AppendLine().
    AppendLine() | Out-Null
$stringBuilder.ToString()
```

Piping output to `Out-Null` is a robust choice for discarding unwanted output. As a pipeline-style operation, values are not held in memory while the pipeline runs.

Alternatives to `Out-Null` include assigning output to the `$null` variable.

Assigning to null

Assigning a statement to the `$null` variable is a popular way of dropping unwanted output. It has the advantage of being obvious, in that it appears at the beginning of the statement:

```
using namespace System.Text
$service = Get-Service | Select-Object -First 1
$stringBuilder = [StringBuilder]::new()
$null = $stringBuilder.
    AppendFormat('Name: {0}', $service.Name).
    AppendLine().
    AppendFormat('Status: {0}', $service.Status).
    AppendLine().
    AppendLine()
$stringBuilder.ToString()
```

Assigning output to `$null` is reasonably fast in all versions of PowerShell. The downside when assigning to `$null` is that memory is allocated to the value on the right-hand side of the assignment; it is only released when the assignment completes.

Redirecting to null

Redirection to `$null` can be added at the end of a statement to discard output. An example of redirection is shown here:

```
using namespace System.Text
$service = Get-Service | Select-Object -First 1
$stringBuilder = [StringBuilder]::new()
$stringBuilder.
    AppendFormat('Name: {0}', $service.Name).AppendLine().
    AppendFormat('Status: {0}', $service.Status).AppendLine().
    AppendLine() > $null
$stringBuilder.ToString()
```

Redirection to `$null` is a relatively popular approach. Like assignment to `$null`, it will assign memory for an object before redirecting output.

Casting to Void

It is possible to cast to `System.Void` to discard output. When using the `StringBuilder` example, this is a clean approach:

```
using namespace System.Text
$stringBuilder = [StringBuilder]::new()
[Void]$stringBuilder.
    AppendFormat('Name: {0}', $service.Name).AppendLine().
    AppendFormat('Status: {0}', $service.Status).AppendLine().
    AppendLine()
$stringBuilder.ToString()
```

However, when used with a command, it requires the use of extra parentheses, which can make the option less appealing. This example uses `Void` to suppress the output from the `Get-Command` command:

```
[Void](Get-Command Get-Command)
```

Like assignment and redirection, casting cannot take place until the command in the value being cast is completely assembled. Memory will be consumed by the statement until the cast is complete.

.NET garbage collection must also run for the consumed memory to be finally released. Although it should not be necessary, garbage collection can be run immediately using the following statement:

```
[GC]::Collect()
```

Any of the preceding methods of disposing of unwanted output used are viable, although in PowerShell 6 and 7, `Out-Null` is the better approach. Personal preference often dictates which to use, especially for code intended to run in both Windows PowerShell and PowerShell 6 or 7.

Working with long lines

There are several techniques you can use when writing scripts to avoid excessively long lines of code. The goal is to avoid needing to scroll to the right when reviewing code. A secondary goal is to avoid littering a script with the backtick (grave accent) character, `` ` ``.

Adding extra line breaks is often a balancing act. Both too many and too few can make it harder to read a script.

The first chapter of this book introduced you to splatting as a means of dealing with commands that require more than a couple of parameters. It remains an important technique for avoiding excessively long lines.

Line break after a pipe

The most obvious technique is perhaps to add a line break after a pipe; for example:

```
Get-Process |
    Where-Object Name -match 'po?w(er)?sh(ell)?'
```

This is useful for long pipelines but may be considered unnecessary for short pipelines. For example, the following short pipeline ends with `ForEach-Object`. The statement is not necessarily long enough to need extra line breaks:

```
Get-Service | Where-Object Status -eq Running | ForEach-Object {
    # Do work on the service
}
```

A pipe is only one of many opportunities to add a line break without needing an escape character.

Line break after an operator

It is possible to add a line break after any of the operators. The most useful place for a line break is often after a logic operator is used to combine several comparisons; for example:

```
Get-Service | Where-Object {
    $_.Status -eq 'Running' -and
    $_.StartType -eq 'Manual'
}
```

One of the less obvious operators is the property dereference operator, "`.`". A line break may be added after calling a method of accessing a property. This is shown in the following example:

```
{ A long string in a script block }.
    ToString().
    SubString(0, 15).
    Trim().
    Length
```

It is not entirely common to find long chains of method calls within PowerShell. A more common scenario makes use of the array operator.

Using the array operator to break up lines

The array operator, `@()`, can break up arrays that are used as arguments into operators, or values for parameters.

For example, the format operator, `-f`, may be used in place of sub-expressions or variable interpolation. `@()` may be used to define an array to hold the arguments for the operator. The following example shows two different ways of creating the same string:

```
$item = Get-Item C:\Windows\explorer.exe
# Sub-expressions and variable interpolation
"The file, $($item.Name), with path $item was last written on $($item.LastWriteTime)"
# The format operator
'The file, {0}, with path {1} was last written on {2}' -f @(
    $item.Name
    $item
    $item.LastWriteTime
)
```

The same approach may be used for `-replace` operations that use particularly long regular expressions. For example, the following `-replace` operation attempts to apply a standard format to a UK telephone number. The regular expression benefits from being on a new line:

```
$ukPhoneNumbers = '+442012345678', '0044(0)1234345678', '+44 (0) 20 81234567', '01234 456789'
$ukPhoneNumbers -replace @(
    '^(?:(?:\+|00)\d{2})?[ -]*(?:\(?0\)?[ -]*)?([138]\d{1,3}|20)[ -]*(\d{3,4})[ -]*(\d{3,4})$'
    '+44 $1 $2 $3'
)
```

The preceding command converts each of the phone numbers into a consistent format:

```
+44 20 1234 5678
+44 1234 345 678
+44 20 8123 4567
+44 1234 456 789
```

In the previous example, the regular expression itself is still extremely long. The approach may be extended further still to break up a complex regular expression:

```
$ukPhoneNumbers = '+442012345678', '0044(0)1234345678', '+44 (0) 20 81234567', '01234 456789'
$ukPhoneNumbers -replace @(
    -join @(
        '^(?:(?:\+|00)\d{2})?[ -]*' # Country code
        '(?:\(?0\)?[ -]*)?'         # Area code prefix
        '([138]\d{1,3}|20)[ -]*'    # Area code
        '(\d{3,4})[ -]*(\d{3,4})$'  # Number
    )
    '+44 $1 $2 $3'
)
```

`@()` may also be used with arguments for commands, such as `Select-Object`:

```
Get-NetAdapter | Select-Object @(
    'Name'
    'Status'
    'MacAddress'
    'LinkSpeed'
    @{ Name = 'IPAddress'; Expression = {
        ($_ | Get-NetIPAddress).IPAddress
    }}
)
```

It is possible to add more line breaks to the Hashtable that describes the `IPAddress` property in the preceding example. Doing so may be beneficial if the `Expression` script is more complex.

Comment-based help

Comment-based help was introduced with PowerShell 2 and allows the author of a script or function to provide content for `Get-Help` without needing to understand and work with more complex MAML help files.

PowerShell includes help for authoring comment-based help:

```
Get-Help about_Comment_Based_Help
```

Comment-based help uses a series of keywords that match up to the different help sections. The following list shows the most used ones:

- `.SYNOPSIS`
- `.DESCRIPTION`
- `.PARAMETER <Name>`

- `.EXAMPLE`
- `.INPUTS`
- `.OUTPUTS`
- `.NOTES`
- `.LINK`

`.SYNOPSIS` and `.DESCRIPTION` are mandatory when writing help. Each of the other sections is optional.

`.PARAMETER`, followed by the name of a parameter, will be included once for each parameter.

`.EXAMPLE` may be used more than once, describing as many examples as desired.

The tag names are not case-sensitive; uppercase is shown here as it is one of the most widely adopted practices. Spelling mistakes in these section names may prevent help from appearing altogether; it is important to be careful when writing comment-based help.

Comment-based help may be used with scripts, functions, and filters and is most often placed first before the `param` block in a script. The synopsis and description sections are shown in the following example:

```
<#
.SYNOPSIS
Briefly describes the main action performed by script.ps1
.DESCRIPTION
A detailed description of the activities of script.ps1.
#>
```

In a function, help is often written inside the body of the function, before the `param` block:

```
function Get-Something {
    <#
    .SYNOPSIS
    Briefly describes the main action performed by Get-Something
    .DESCRIPTION
    A detailed description of the activities of Get-Something.
    #>
}
```

Help for individual parameters can be defined in two different ways when using comment-based help.

Parameter help

Parameter help is most often written using the `.PARAMETER` tag, as shown in the following example:

```
function Get-Something {
    <#
    .SYNOPSIS
    Briefly describes the main action performed by Get-Something
    .DESCRIPTION
    A detailed description of the activities of Get-Something.
    .PARAMETER Parameter1
    Describes the purpose of Parameter1.
    .PARAMETER Parameter2
    Describes the purpose of Parameter2.
    #>
    param (
        $Parameter1,
        $Parameter2
    )
}
```

Help for a parameter may also be written as a comment above the parameter:

```
function Get-Something {
    <#
    .SYNOPSIS
    Briefly describes the main action performed by Get-Something
    .DESCRIPTION
    A detailed description of the activities of Get-Something.
    #>
    param (
        # Describes the purpose of Parameter1.
        $Parameter1,
        # Describes the purpose of Parameter2.
        $Parameter2
    )
}
```

One possible advantage of this approach is that it is easy to see which parameters have help and which do not.

Regardless of where help is written for a parameter, `Get-Help` will read the value:

```
PS> Get-Help Get-Something -Parameter Parameter1
-Parameter1 <Object>
    Describes the purpose of Parameter1.
    Required? false
    Position? 1
    Default value
    Accept pipeline input? false
    Accept wildcard characters? false
```

Examples of how to use a script or function are helpful, and more than one example can be added.

Examples

`Get-Help` expects examples to start with one or more lines of code, followed by a description of the example; for example:

```
function Get-Something {
    <#
    .SYNOPSIS
    Briefly describes the main action performed by Get-Something
    .DESCRIPTION
    A detailed description of the activities of Get-Something.
    .EXAMPLE
    $something = Get-Something
    $something | Do-Something
    Gets something from somewhere.
    #>
    param (
        # Describes the purpose of Parameter1.
        $Parameter1,
        # Describes the purpose of Parameter2.
        $Parameter2
    )
}
```

The help parser is quite simple when it comes to comment-based help. Only the very first line of an example is code. This can be demonstrated by exploring the object returned by `Get-Help` based on the preceding example:

```
PS> (Get-Help Get-Something -Examples).examples[0].example.code
$something = Get-Something
```

The rest of the code is part of the remark. An open pull request strives to improve this part of the help parser; it may improve in time.

Summary

This chapter introduced writing scripts and functions, including brief guidance on establishing a style, followed by an exploration of the small differences between scripts, functions, and script blocks.

You can use parameters to accept user input for scripts, functions, and script blocks. The `param` block can be used to define the list of parameters.

Named blocks are used when acting on pipeline input. Each block executes at a different point in the pipeline lifecycle. The function and filter keywords use a different default named block, but otherwise have identical functionality. The `begin` block in all commands in a pipeline executes before a pipeline starts, the `process` block executes once for each value passed from one function to another, and the `end` block executes once for each function after the last pipeline value is passed.

The `cleanup` block was very briefly introduced as an up-and-coming feature, hopefully one that will make it into PowerShell 7 soon.

Output must be managed when writing code; it is important to appreciate that all statements that can generate output will send values to the output pipeline unless action is taken to prevent this. The `return` keyword was briefly explored as a means of ending a function or script early.

It is sometimes difficult, when writing code, to make a long statement easy to read. Several strategies for breaking up long lines of code were explored, such as using the array operator, or adding a line break after an operator.

The final section covered the structure of commend-based help, including parameter help and examples. Documentation plays a vital role in writing maintainable code.

The next chapter, *Chapter 18, Parameters, Validation, and Dynamic Parameters*, builds on this chapter, exploring the extensive capabilities of the `param` block in much greater detail.

18

Parameters, Validation, and Dynamic Parameters

PowerShell has an extensive parameter handling and validation system that you can use in scripts and functions. The system allows a developer to make parameters mandatory; to define what, if any, positional binding is allowed; to fill parameters from the pipeline; to describe different parameter sets; and to validate the values passed to a parameter. The wealth of options available makes parameter handling an incredibly involved subject.

This chapter explores the following topics:

- The `Parameter` attribute
- Validating input
- Pipeline input
- Defining parameter sets
- Argument completers
- Dynamic parameters

Parameters in PowerShell can make use of the `Parameter` attribute to define how the parameter should behave.

The Parameter attribute

The `Parameter` attribute is an optional attribute that you can use to define some of the behavior of a parameter, such as making a parameter mandatory.

Creating an instance of the `Parameter` object shows the different properties that you can set:

```
PS> [Parameter]::new()
ExperimentName                  :
ExperimentAction                : None
Position                        : -2147483648
ParameterSetName                : __AllParameterSets
Mandatory                       : False
ValueFromPipeline               : False
ValueFromPipelineByPropertyName : False
ValueFromRemainingArguments     : False
HelpMessage                     :
HelpMessageBaseName             :
HelpMessageResourceId           :
DontShow                        : False
TypeId                          : System.Management.Automation.Pa…
```

A few of these properties should be ignored as they are not intended to be set directly, such as `HelpMessageBaseName`, `HelpMessageResourceId`, and `TypeId`.

`ExperimentName` and `ExperimentAction` are for use with experimental features in PowerShell 7; the properties are not available in Windows PowerShell. These properties allow parameters to be made available on commands if a corresponding experimental feature is enabled. These properties do not appear to be in use at the time of writing.

Several other complex properties are explored in other sections in this chapter, such as `ParameterSetName`, `ValueFromPipeline`, and `ValueFromPipelineByPropertyName`.

The `Parameter` attribute is placed before the parameter itself. The following example shows the simplest use of the `Parameter` attribute:

```
[CmdletBinding()]
param (
    [Parameter()]
    $Parameter1
)
```

Using the `Parameter` attribute has the side effect of turning a basic function or script into an advanced function or script, even when the `CmdletBinding` attribute itself is missing. You can use `Get-Command` to explore whether `CmdletBinding` is present:

```
PS> function Test-CmdletBinding {
>>     param (
>>         [Parameter()]
>>         $Parameter1
>>     )
>> }
PS> Get-Command Test-CmdletBinding | Select-Object CmdletBinding
CmdletBinding
-------------
         True
```

This means that the common parameters, including `Verbose` and `ErrorAction`, are available to any function that uses the `Parameter` attribute (for any parameter).

Starting with PowerShell 3, Boolean properties, such as `Mandatory` and `ValueFromPipeline`, may be written without providing an expression. For example, `Parameter1` is made mandatory in the following code:

```
function Test-Mandatory {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory)]
        $Parameter1
    )
}
```

Use of `Mandatory = $false`

The default value for the `Mandatory` property is `false`; setting an explicit value of default may reduce readability.

`Mandatory` is significant and stands out, but the value assigned is less significant and, when reading rapidly, it might be assumed to be `true` simply because the property is present.

Position and positional binding

Position defaults to `-2147483648`, the smallest possible value for `Int32` (see `[Int32]::MinValue`). Unless an explicit permission is set, parameters may be bound in the order they are written in the parameter block. Setting the `PositionalBinding` property of `CmdletBinding` to false can disable this behavior.

Automatic positional binding is shown in the following example:

```
function Test-Position {
    [CmdletBinding()]
    param (
        [Parameter()]
        $Parameter1,
        [Parameter()]
        $Parameter2
    )
    '{0}-{1}' -f $Parameter1, $Parameter2
}
```

When called, the command shows that `Parameter1` and `Parameter2` have been filled with the values supplied using position only:

```
PS> Test-Position 1 2
1-2
```

Automatic positional binding is available by default; the `Parameter` attribute is not required. An explicit definition of position allows greater control and effectively disables automatic positional binding:

```
function Test-Position {
     param (
        [Parameter(Position = 1)]
        $Parameter1,
        $Parameter2
    )
}
```

Automatic positional binding is disabled when using parameter sets. Parameter sets are explored later in this chapter.

Exploring command metadata shows that positional binding is still enabled, but as this is an ordered operation, the default position no longer has meaning. The command metadata shows that positional binding is still enabled:

```
PS> [System.Management.Automation.CommandMetadata](
>>     Get-Command Test-Position
>> )
Name                    : Test-Position
CommandType             :
DefaultParameterSetName :
SupportsShouldProcess   : False
SupportsPaging          : False
PositionalBinding       : True
SupportsTransactions    : False
HelpUri                 :
RemotingCapability      : PowerShell
ConfirmImpact           : Medium
Parameters              : {[Parameter1, System.Management.Automation.ParameterMetadata], [Parameter2,
System.Management.Automation.ParameterMetadata]}
```

Attempting to pass a value for `Parameter2` by position will raise an error:

```
PS> Test-Position 1 2
Test-Position: A positional parameter cannot be found that accepts argument '2'.
```

PowerShell orders parameters based on the position value. The value must be greater than `-2147483648`. It is possible, but not advisable, to set `Position` to a negative value. The accepted practice has numbering starting at either `0` or `1`.

The DontShow property

You can use `DontShow` to hide a parameter from tab completion and IntelliSense. `DontShow` is rarely used but may occasionally be useful for short recursive functions. The following function recursively calls itself, comparing `MaxDepth` and `CurrentDepth`. The `CurrentDepth` parameter is owned by the function and a user is never expected to supply a value:

```
function Show-Property {
    [CmdletBinding()]
    param (
        # Show the properties of the specified object.
        [Parameter(Mandatory)]
        [PSObject]$InputObject,
        # The maximum depth when expanding properties
        # of child objects.
        [Int32]$MaxDepth = 5,
        # Used to track the current depth during recursion.
        [Parameter(DontShow)]
        [Int32]$CurrentDepth = 0
    )
    $width = $InputObject.PSObject.Properties.Name |
        Sort-Object { $_.Length } -Descending |
        Select-Object -First 1 -ExpandProperty Length
    foreach ($property in $InputObject.PSObject.Properties) {
        '{0}{1}: {2}' -f @(
            ' ' * $CurrentDepth
            $property.Name.PadRight($width, ' ')
            $property.TypeNameOfValue
        )
        if ($CurrentDepth -lt $MaxDepth -and
            $property.Value -and
            -not $property.TypeNameOfValue.IsPrimitive) {
            $params = @{
                InputObject = $property.Value
                CurrentDepth = $CurrentDepth + 1
            }
            Show-Property @params
        }
    }
}
```

Marking a parameter as `DontShow` hides the parameter to a degree, but it does nothing to prevent a user from providing a value for the parameter. In this preceding case, a better approach might be to move the body of the function into a nested function. Alternatively, if the function is part of a module, the recursive code might be moved to a function that is not exported from a module and exposed by a second, tidier, function.

The ValueFromRemainingArguments property

Setting the `ValueFromRemainingArguments` property allows a parameter to consume all the other arguments supplied for a command. You can use this to make an advanced function act similarly to a basic function.

For example, this basic function will fill the `Parameter1` parameter with the first argument and ignore all others. The extra values are added to the `$args` automatic variable and are listed in the `UnboundArguments` property of the `$MyInvocation` automatic variable:

```
function Test-BasicBinding {
    param (
        $Parameter1
    )
    $MyInvocation.UnboundArguments
}
```

Calling the function with non-existent parameters will not raise an error. The additional values are added to the `UnboundArguments` array (and the `$args` variable):

```
PS> Test-BasicBinding -Parameter1 value1 -Parameter2 value2
-Parameter2
Value2
```

Without a declared parameter in the `param` block, `Parameter2` is just another value, it is not parsed as the name of a parameter. You can use the `ValueFromRemainingArguments` property to make an advanced function behave in much the same way as the preceding basic function:

```
function Test-AdvancedBinding {
    [CmdletBinding()]
    param (
        $Parameter1,
        [Parameter(ValueFromRemainingArguments)]
        $OtherArguments
    )
    $OtherArguments
}
```

If the `$OtherArguments` parameter is not for the normal use of the function, the `DontShow` property might be added to make it less obvious and intrusive.

The HelpMessage property

`HelpMessage` is only applied to `Mandatory` parameters and is not particularly useful. If a parameter is mandatory and is not passed when a command is called, a prompt for the parameter value will appear. Typing `!?` in the prompt, instead of a value, will display the help message text:

```
PS> function Test-HelpMessage {
>>     param (
>>         [Parameter(
>>             Mandatory,
>>             HelpMessage = 'Help text for Parameter1'
>>         )]
>>         $Parameter1
>>     )
>> }
PS> Test-HelpMessage
cmdlet Test-HelpMessage at command pipeline position 1
Supply values for the following parameters:
(Type !? for Help.)
Parameter1: !?
Help text for Parameter1
Parameter1:
```

Given that `HelpMessage` is only visible when explicitly requested in this manner, it is most often ignored entirely. It is better to spend time writing help content for a parameter than writing values for `HelpMessage`.

Beyond making `Parameter` mandatory, the `Parameter` attribute does not control the values that can be used with a parameter. PowerShell offers attributes for validating parameter arguments.

Validating input

PowerShell provides a variety of different ways to tightly define the content for a parameter. Assigning a .NET type to a parameter is the first of these. If a parameter is set as `[String]`, it will only ever hold a value of that type. PowerShell will attempt to coerce any values passed to the parameter into that type.

The PSTypeName attribute

The `PSTypeName` attribute can test the type name assigned to a custom object.

It is not uncommon in PowerShell to want to pass an object created in one command, as a `PSObject` (or `PSCustomObject`), to another.

Type names are assigned by setting (or adding) a value to the hidden `PSTypeName` property. There are several ways to tag `PSCustomObject` with a type name.

The simplest is to set a value for a `PSTypeName` property, shown as follows:

```
$object = [PSCustomObject]@{
    Property   = 'Value'
    PSTypeName = 'SomeTypeName'
}
```

The `PSTypeName` property does not exist on the resulting object, but `Get-Member` will now show the new type name:

```
PS> $object | Get-Member
TypeName: SomeTypeName
Name            MemberType      Definition
----            ----------      ----------
Equals          Method          bool Equals(System.Object obj)
GetHashCode     Method          int GetHashCode()
GetType         Method          type GetType()
ToString        Method          string ToString()
Property        NoteProperty    string Property=Value
```

The type name will show at the top of the hidden `PSTypeNames` property:

```
PS> $object.PSTypeNames
SomeTypeName
System.Management.Automation.PSCustomObject
System.Object
```

It is also possible to add to the `PSTypeNames` array directly:

```
$object = [PSCustomObject]@{ Property = 'Value' }
# Add to the end of the existing list
$object.PSTypeNames.Add('SomeTypeName')
# Or add to the beginning of the list
$object.PSTypeNames.Insert(0, 'SomeTypeName')
```

Finally, `Add-Member` can add to `PSTypeNames`. If used, it adds the new type name at the top of the existing list:

```
$object = [PSCustomObject]@{ Property = 'Value' }
$object | Add-Member -TypeName 'SomeTypeName'
```

These tagged types may be tested using the `PSTypeName` attribute of a parameter, for example:

```
function Test-PSTypeName {
    [CmdletBinding()]
    param (
        [PSTypeName('SomeTypeName')]
        $InputObject
    )
}
```

This technique is used by many of the WMI-based commands implemented by Microsoft. For example, the `Set-NetAdapter` command uses a `PSTypeName` attribute for its `InputObject` parameter:

```
$command = Get-Command Set-NetAdapter
$command.Parameters['InputObject'].Attributes |
    Where-Object  TypeId -eq ([PSTypeNameAttribute])
```

In the case of the WMI-based commands, this is used in addition to a .NET type name, an array of `CimInstance`. This type of parameter is like the following example:

```
function Test-PSTypeName {
    [CmdletBinding()]
    param (
        [Parameter(
            Mandatory,
            ValueFromPipeline,
            ParameterSetName = 'InputObject (cdxml)'
        )]
        [PSTypeName('Microsoft.Management.Infrastructure.CimInstance#MSFT_NetAdapter')]
        [CimInstance[]]$InputObject
    )
}
```

This technique is incredibly useful when the .NET object type is not sufficiently detailed to restrict input, as is the case with the `CimInstance` type just used. This is true of `PSCustomObject` input as much as the `CimInstance` array type used before.

Validation attributes

PowerShell offers several validation attributes to test the content of arguments passed to parameters. There are two general classes of validation attribute; the first validates the argument as a single object, which tests the value as a whole:

- `ValidateNotNull`
- `ValidateNotNullOrEmpty`
- `ValidateCount`
- `ValidateDrive`
- `ValidateUserDrive`

The second validates enumerated arguments. These validation attributes can be applied to parameters that accept arrays. The validation step applies to each element in the array. The enumerated argument validation attributes are:

- `ValidateLength`
- `ValidatePattern`
- `ValidateRange`
- `ValidateScript`
- `ValidateSet`

The validation attributes are documented in `about_Functions_Advanced_Parameters`, except for the newer `ValidateDrive` attribute, which was introduced with PowerShell 5. The online version of the about document is available at: https://docs.microsoft.com/powershell/module/microsoft.powershell.core/about/about_functions_advanced_parameters.

The constructor for a validation attribute can be explored to determine the arguments it supports. This may be discovered using the `::new` static method in PowerShell 5 and later, for example, `ValidateDrive`:

```
PS> [ValidateDrive]::new
OverloadDefinitions
-------------------
ValidateDrive new(Params string[] validRootDrives)
```

Each of the available attributes, including `ValidateDrive`, is demonstrated as follows.

The ValidateNotNull attribute

`ValidateNotNull` may be used with parameters that are not flagged as mandatory. It is applicable where an object type can accept null input. Such types include object, `CimInstance`, and array types. The following is the simplest example of `ValidateNotNull`:

```
function Test-ValidateNotNull {
    [CmdletBinding()]
    param (
        [ValidateNotNull()]
        $Parameter1
    )
}
```

As `Parameter1` is, by default, set to the `Object` type, it would ordinarily accept a null value. When applied to an array, it prohibits null values but retains the ability to pass an empty array into a function, for example:

```
function Test-ValidateNotNull {
    [CmdletBinding()]
    param (
        [ValidateNotNull()]
        [String[]]$Parameter1
    )
}
```

If a null value is explicitly passed, an error will be raised:

```
PS> Test-ValidateNotNull -Parameter1 $null
Test-ValidateNotNull: Cannot validate argument on parameter 'Parameter1'. The argument is null.
Provide a valid value for the argument, and then try running the command again.
```

The `ValidateNotNull` attribute has no effect on `String` or numeric types (such as `Byte`, `Int`, or `Int64`).

The ValidateNotNullOrEmpty attribute

`ValidateNotNullOrEmpty` extends `ValidateNotNull` to disallow empty arrays and empty strings:

```
function Test-ValidateNotNullOrEmpty {
    [CmdletBinding()]
    param (
        [ValidateNotNullOrEmpty()]
        [String]$Parameter1,
        [ValidateNotNullOrEmpty()]
        [Object[]]$Parameter2
    )
}
```

An error will be thrown if either an empty string is supplied for `Parameter1`, or an empty array is supplied for `Parameter2`. Like `ValidateNotNull`, `ValidateNotNullOrEmpty` has no effect on numeric types.

The ValidateCount attribute

You can use `ValidateCount` to test the size of an array supplied to a parameter. The attribute expects a minimum and maximum length for the array.

`ValidateCount` only has meaning when applied to an array-type parameter, for example:

```
function Test-ValidateCount {
    [CmdletBinding()]
    param (
        [ValidateCount(1, 1)]
        [String[]]$Parameter1
    )
}
```

You can also apply `ValidateCount` to parameters that accept collection types such as `System.Collections.ArrayList` or `System.Collections.Generic.List`.

The ValidateDrive attribute

`ValidateDrive` may be used to test the drive letter provided for a parameter that accepts a path. `ValidateDrive` handles both relative and absolute paths. A relative path is resolved to a full path before it is tested against the supplied drive letters. When using the `ValidateDrive` attribute, the parameter type must be `String`. The parameter cannot be omitted:

```
function Test-ValidateDrive {
    [CmdletBinding()]
    param (
        [ValidateDrive('C')]
        [String]$Parameter1
    )
}
```

`ValidateDrive` cannot act on an array of paths; if the parameter type is an array, an error will be thrown stating that the path argument is invalid.

The ValidateUserDrive attribute

The `ValidateUserDrive` attribute is written for use within **Just Enough Administration** (**JEA**) configurations. The attribute validates that the specified drive is the `User` drive, as specified in a session configuration.

Just Enough Administration, or JEA, was briefly explored in *Chapter 14*, *Remoting and Remote Management*. The `User` drive configuration is described on Microsoft Docs: https://docs.microsoft.com/powershell/scripting/learn/remoting/jea/session-configurations.

It is used to validate that the root drive for a path is the JEA User Drive:

```
function Test-ValidateDrive {
    [CmdletBinding()]
    param (
        [ValidateUserDrive()]
        [String]$Parameter1
    )
}
```

The `ValidateUserDrive` attribute is equivalent to using `ValidateDrive` with `User` as an argument:

```
[ValidateDrive('User')]
[String]$Parameter1
```

As noted above, the definition of a `User` drive is part of the JEA session configuration.

The ValidateLength attribute

`ValidateLength` can be applied to a `String` parameter or a parameter that contains an array of strings. Each string will be tested against the minimum and maximum length:

```
function Test-ValidateLength {
    [CmdletBinding()]
    param (
        [ValidateLength(2, 6)]
        [String[]]$Parameter1
    )
}
```

Any string with a length below the minimum or above the maximum will trigger an error.

The ValidatePattern attribute

You use `ValidatePattern` to test that a string, or the elements in an array of strings, matches the supplied pattern:

```
function Test-ValidatePattern {
    [CmdletBinding()]
    param (
        [ValidatePattern('^Hello')]
        [String]$Parameter1
    )
}
```

In addition to the pattern argument, `ValidatePattern` accepts `RegexOptions` using the `Options` named parameter, for example:

```
function Test-ValidatePattern {
    [CmdletBinding()]
    param (
        [ValidatePattern('^Hello', Options = 'Multiline')]
        [String]$Parameter1
    )
}
```

The possible values for `Options` are described by the `System.Text.RegularExpressions.RegexOptions` enumeration, which is documented on Microsoft Docs:
https://docs.microsoft.com/dotnet/api/system.text.regularexpressions.regexoptions.

Further information on each option is available in the *Regular Expression Options* document:
https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-options.

You can include multiple options as a comma-separated list, for example:

```
[ValidatePattern('^Hello', Options = 'IgnoreCase, Multiline')]
```

By default, the `IgnoreCase` option is set. If you want to make a pattern case-sensitive, `Options` can be set to `None`:

```
[ValidatePattern('^Hello', Options = 'None')]
```

A criticism that might be leveled against `ValidatePattern` is that there is no way to customize or define the error message in Windows PowerShell.

PowerShell 6 and above adds an optional `ErrorMessage` parameter to tackle this problem. The default error message written by `ValidatePattern` is shown as follows:

```
function Test-ValidatePattern {
    [CmdletBinding()]
    param (
        [ValidatePattern(
            '^[A-Z]',
            Options = 'None'
        )]
        [String]$Greeting
    )
}
```

The error message generated by the attribute is shown here:

```
PS> Test-ValidatePattern -Greeting 'hello Jim.'
Test-ValidatePattern: Cannot validate argument on parameter 'Greeting'. The argument "hello Jim." does
not match the "^[A-Z]" pattern. Supply an argument that matches "^[A-Z]" and try the command again.
```

You may see an alternative message in PowerShell 6 and above:

```
function Test-ValidatePattern {
    [CmdletBinding()]
    param (
        [ValidatePattern(
           '^[A-Z]',
           Options      = 'None',
           ErrorMessage = 'Must start with a capital letter.'
        )]
        [String]$Greeting
    )
}
```

Running the new command with an invalid string shows the new error message:

```
PS> Test-ValidatePattern -Greeting 'hello Jim.'
Test-ValidatePattern: Cannot validate argument on parameter 'Greeting'. Must start with a capital
letter.
```

The new error message is far nicer than showing an end user a regular expression, even a relatively simple regular expression.

The ValidateRange attribute

`ValidateRange` tests whether a value, or an array of values, fall within a specified range. `ValidateRange` is most used to test numeric ranges. However, it is also able to test strings. For example, the "z" string can be said to be within the "A" to "Z" range. This approach is slightly harder to apply when testing strings as the "Zz" string is greater than "Z".

The following example uses `ValidateRange` to test an integer value:

```
function Test-ValidateRange {
    [CmdletBinding()]
    param (
        [ValidateRange(1, 20)]
        [Int]$Parameter1
    )
}
```

The ValidateScript attribute

`ValidateScript` executes an arbitrary block of code against each of the arguments for a parameter. If the argument is an array, each element is tested. One common use for `ValidateScript` is to test whether a path exists, for example:

```
function Test-ValidateScript {
    [CmdletBinding()]
    param (
        [ValidateScript( { Test-Path $_ -PathType Leaf } )]
        [String]$Path
    )
}
```

`ValidateScript` can contain just about anything a developer desires, although it is generally recommended to keep validation scripts short.

In PowerShell 6 and above, `ValidateScript` gains an optional `ErrorMessage` parameter that replaces the default message, which repeats the failed script to the end user:

```
function Test-ValidateScript {
    [CmdletBinding()]
    param (
        [ValidateScript(
            { Test-Path $_ -PathType Leaf },
            ErrorMessage = 'Path must be an existing file'
        )]
        [String]$Path
    )
}
```

In Windows PowerShell, you can use `throw` within the script to write a more friendly error message at the cost of a more complex script:

```
function Test-ValidateScript {
    [CmdletBinding()]
    param (
        [ValidateScript({
            if (Test-Path $_ -PathType Leaf) {
                $true
            } else {
                throw 'Path must be an existing file'
            }
        })]
        [String]$Path
    )
}
```

The ValidateSet attribute

`ValidateSet` tests whether the specified argument, or each of an array of arguments, exists in a set of possible values:

```
function Test-ValidateSet {
    [CmdletBinding()]
    param (
        [ValidateSet('One', 'Two', 'Three')]
        [String]$Value
    )
}
```

The set of values must be hardcoded in the attribute and cannot be derived from a variable or another command. By default, the set is not case-sensitive. If desirable, you can make the set case-sensitive by using the `IgnoreCase` named parameter:

```
function Test-ValidateSet {
    [CmdletBinding()]
    param (
        [ValidateSet('One', 'Two', 'Three', IgnoreCase = $false)]
        [String]$Value
    )
}
```

Like `ValidatePattern` and `ValidateSet`, `ValidateSet` gains an optional `ErrorMessage` parameter in PowerShell 6 and above.

PowerShell 6 also added the ability to dynamically define the values that `ValidateSet` can use. The values are defined in a class that implements a `GetValidValues` method. An example of this class is available in the next chapter, *Chapter 19, Classes and Enumerations*.

The Allow attributes

The `Allow` attributes are most frequently used when a parameter is mandatory. If a parameter is mandatory, PowerShell will automatically disallow the assignment of empty values, that is, empty strings and empty arrays. The `Allow` attributes can be used to modify that behavior. The attributes make it possible to require a parameter, but still allow empty values.

The AllowNull attribute

`AllowNull` is used to permit explicit use of `$null` as a value for a `Mandatory` parameter:

```
function Test-AllowNull {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory)]
        [AllowNull()]
        [Object]$Parameter1
    )
}
```

`AllowNull` is effective for array parameters, and for parameters that use `Object` as a type. `AllowNull` is not effective for string parameters as the null value is cast to an empty string, and an empty string is still not permitted.

The AllowEmptyString attribute

`AllowEmptyString` fills the gap, allowing both null and empty values to be supplied for a mandatory string parameter. In both cases, the resulting assignment will be an empty string. It is not possible to distinguish between a value passed as null and a value passed as an empty string:

```
function Test-AllowEmptyString {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory)]
        [AllowEmptyString()]
        [String]$Parameter1
    )
}
```

The AllowEmptyCollection attribute

`AllowEmptyCollection`, as the name suggests, allows an empty array to be assigned to a mandatory parameter:

```
function Test-AllowEmptyCollection {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory)]
        [AllowEmptyCollection()]
        [Object[]]$Parameter1
    )
}
```

This will allow the command to be called with an explicitly empty array:

```
Test-AllowEmptyCollection -Parameter1 @()
```

PSReference parameters

Many of the object types used in PowerShell are reference types. When an object is passed to a function, any changes made to that object will be visible outside the function, irrespective of the output generated by the command. For example, the following function accepts an object as input and then changes the value of a property on that object:

```
function Set-Value {
    [CmdletBinding()]
    param (
        [PSObject]$Object
    )
    $Object.Value = 2
}
```

When the function is passed an object, the change can be seen on any other variables that reference that object:

```
PS> $myObject = [PSCustomObject]@{ Value = 1 }
PS> Set-Value $myObject
PS> $myObject.Value
2
```

Strings, numeric values, and dates, on the other hand, are all examples of value types. Changes made to a value type inside a function will not be reflected in variables that reference that value elsewhere; a new value is created.

Occasionally, it is desirable to make a function affect the content of a value type without either returning the value as output or changing the value of a property of an object. The `PSReference` type, `[Ref]`, can be used to achieve this. The following function normally returns `True` or `False` depending on whether `Get-Date` successfully parsed the date string into a `DateTime` object:

```
function Test-Date {
    [CmdletBinding()]
    param (
        [String]$Date,
        [Ref]$DateTime
    )
    if ($value = Get-Date $Date -ErrorAction SilentlyContinue) {
        if ($DateTime) {
            $DateTime.Value = $value
        }
        $true
    } else {
        $false
    }
}
```

When the function is run, a variable that holds an existing `DateTime` object might be passed as an optional reference. PowerShell can update the date via the reference, changing the value outside of the function:

```
PS> $dateTime = Get-Date
PS> Test-Date 01/01/2019 -DateTime ([Ref]$dateTime)
True
PS> $dateTime
01 January 2019 00:00:00
```

The same behavior can be seen with Boolean, string, and numeric types.

PowerShell offers a wide variety of attributes that help to validate user input without writing bespoke checks in the body of a function.

Moving back to the `Parameter` attribute again, the `Parameter` attribute can be used to define how a parameter behaves in a pipeline.

Pipeline input

Using the `Parameter` attribute to set either `ValueFromPipeline` or `ValueFromPipelineByPropertyName` sets a parameter up to fill from the input pipeline.

The pipeline is a complex topic and requires a background in the use of named blocks. Named blocks, along with a broader set of examples for pipeline usage, are available in *Chapter 17, Scripts, Functions, and Script Blocks*.

About ValueFromPipeline

`ValueFromPipeline` allows the entire object to be passed into a parameter from an input pipeline. The following function implements an `InputObject` parameter, which accepts pipeline input by using the `ValueFromPipeline` property of the `Parameter` attribute:

```
function Get-InputObject {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory, ValueFromPipeline)]
        $InputObject
    )
    process {
        'Input object was of type {0}' -f @(
            $InputObject.GetType().FullName
        )
    }
}
```

Remember that values read from an input pipeline are only available in the `process` block of a script or function. As the default type assigned to a parameter is `Object`, this will accept any kind of input that might be passed. This behaves in a similar manner to the `InputObject` parameter for `Get-Member` or `Select-Object`, for example.

Accepting null input

Commands such as `Where-Object` allow an explicit null value in the input pipeline. To allow null in an input pipeline, the `[AllowNull()]` attribute would be added to the `InputObject` parameter. There is a difference between supporting `$null | Get-InputObject` and implementing pipeline support originating from a command that returns nothing: `AllowNull` is only needed when an explicit null is in the input pipeline.

In the following example, the `Get-EmptyOutput` function has no body and will not return anything. This simulates a command that returns nothing because all the output is filtered out. The `Get-InputObject` function can take part in a pipeline with `Get-`

`EmptyOutput` without using `AllowNull`:

```
function Get-EmptyOutput { }
function Get-InputObject {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory, ValueFromPipeline)]
        $InputObject
    )
}
# No output, no error
Get-EmptyOutput | Get-InputObject
```

If `Get-EmptyOutput` were to explicitly return null, which is not a good practice, `Get-InputObject` would raise a parameter binding error:

```
PS> function Get-EmptyOutput { return $null }
PS> Get-EmptyOutput | Get-InputObject
Get-InputObject: Cannot bind argument to parameter 'InputObject' because it is null.
```

Adding `AllowNull` would sidestep this error, but `Get-InputObject` would have to handle a null value internally:

```
function Get-EmptyOutput { return $null }
function Get-InputObject {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory, ValueFromPipeline)]
        [AllowNull()]
        $InputObject
    )
    if ($InputObject) {
        # Do work
    }
}
# No output, no error
Get-EmptyOutput | Get-InputObject
```

If this were real output from a function, it may be better to consider the output from `Get-EmptyOutput` to be a bug and pass it through `Where-Object` to sanitize the input, which avoids the need to add `AllowNull`, for example:

```
Get-EmptyOutput | Where-Object { $_ } | Get-InputObject
```

Input object types

If a type is defined for the `InputObject` variable, the command will only work if the input pipeline contains that object type. An error will be thrown when a different object type is passed. The following example modifies the command to accept pipeline input from `Get-Process`; it expects objects of the `System.Diagnostics.Process` type only:

```
function Get-InputObject {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory, ValueFromPipeline)]
        [System.Diagnostics.Process]$InputObject
    )
    process {
        'Process name {0}' -f $InputObject.Name
    }
}
```

Attempting to pipe a value that is something other than a `System.Diagnostics.Process` value will result in an error:

```
PS> 1 | Get-InputObject
Get-InputObject: The input object cannot be bound to any parameters for the command either because the
command does not take pipeline input or the input and its properties do not match any of the
parameters that take pipeline input.
```

Using ValueFromPipeline for multiple parameters

If more than one parameter uses `ValueFromPipeline`, PowerShell will attempt to provide values to each. The parameter binder can be said to be greedy in this respect. The following function can be used to show that both parameters are filled with the same value if the parameters accept the same type, or if the value can be coerced into that type:

```
function Test-ValueFromPipeline {
    [CmdletBinding()]
    param (
        [Parameter(ValueFromPipeline)]
        [Int]$Parameter1,
        [Parameter(ValueFromPipeline)]
        [Int]$Parameter2
    )
    process {
        'Parameter1: {0}:: Parameter2: {1}' -f @(
            $Parameter1
            $Parameter2
        )
    }
}
```

Providing an input pipeline for the command shows the values assigned to each parameter:

```
PS> 1..2 | Test-ValueFromPipeline
Parameter1: 1 :: Parameter2: 1
Parameter1: 2 :: Parameter2: 2
```

Filling variables is the job of the parameter binder in PowerShell. Using `Trace-Command` will show the parameter binder in action:

```
Trace-Command -PSHost -Name ParameterBinding -Expression {
    1 | Test-ValueFromPipeline
}
```

The bind-pipeline section will display messages that show that the value was successfully bound to each parameter. If the two parameters expected different types, the parameter binding will attempt to coerce the value into the requested type. If that is not possible, it will give up on the attempt to fill the parameter. The next example declares two different parameters; both accept values from the pipeline and neither is mandatory:

```
function Get-InputObject {
    [CmdletBinding()]
    param (
        [Parameter(ValueFromPipeline)]
        [System.Diagnostics.Process]$ProcessObject,
        [Parameter(ValueFromPipeline)]
        [System.ServiceProcess.ServiceController]$ServiceObject
    )
    process {
        if ($ProcessObject) {
            'Process: {0}' -f $ProcessObject.Name
        }
        if ($ServiceObject) {
            'Service: {0}' -f $ServiceObject.Name
        }
    }
}
```

The command will, at this point, accept pipeline input from both `Get-Process` and `Get-Service`. Each command will fill the matching parameter, `Get-Process` fills `ProcessObject`, and `Get-Service` fills `ServiceObject`.

This design is unusual and perhaps confusing; it is only demonstrated here because it is possible. A parameter set can be used to make sense of the pattern, which is explored in the *Defining parameter sets* section.

Using PSTypeName

You can use the `PSTypeName` attribute to tightly define the acceptable objects for a parameter from a pipeline:

```
function Get-InputObject {
    [CmdletBinding()]
    param (
        [Parameter(ValueFromPipeline)]
        [PSTypeName('CustomTypeName')]
        $InputObject
    )
    process {
        $InputObject.Name
    }
}
```

This function would accept input from an object that declares the matching type name:

```
[PSCustomObject]@{
    Name       = 'Value'
    PSTypeName = 'CustomTypeName'
} | Get-InputObject
```

The more specific the type name, the better. You can use `PSTypeName` to validate .NET type names, but this is typically better handled by assigning a .NET type to the parameter directly.

About ValueFromPipelineByPropertyName

`ValueFromPipelineByPropertyName` attempts to fill a parameter from the property of an object in the input pipeline. When filling a value by property name, the name and type of the property is important, but not the object that implements the property.

For example, you might create a function to accept a string value from a `Name` property:

```
function Get-Name {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory, ValueFromPipelineByPropertyName)]
        [String]$Name
    )
    process {
        $Name
    }
}
```

Any command that returns an object that contains a `Name` property in a string is acceptable input for this function. Additional parameters might be defined, which would further restrict the input object type, assuming the new properties are mandatory:

```
function Get-Status {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory, ValueFromPipelineByPropertyName)]
        [String]$Name,
        [Parameter(Mandatory, ValueFromPipelineByPropertyName)]
        [String]$Status
    )
    process {
        '{0}: {1}' -f $Name, $Status
    }
}
```

This new function would accept pipeline input from `Get-Service`, as the output from `Get-Service` has both `Name` and `Status` properties. Using `Get-Member` against `Get-Service` would show that the `Status` property is an enumeration value described by `System.ServiceProcess.ServiceControllerStatus`. This value is acceptable to the `Get-Status` function as it can be coerced into a string, which satisfies the `Status` parameter.

The previous function is not limited to a specific input object type. A `PSCustomObject` can be created with properties to satisfy the parameters for the `Get-Status` function:

```
[PSCustomObject]@{ Name = 'Name'; Status = 'Running' } |
    Get-Status
```

As with the `ValueFromPipeline` input, the parameter binder will attempt to fill as many of the parameters as possible from the input pipeline. `Trace-Command`, as used when exploring `ValueFromPipeline`, shows the behavior of the parameter binder.

ValueFromPipelineByPropertyName and parameter aliases

Any parameter may be given one or more aliases using the `Alias` attribute, as shown in the following example:

```
function Get-InputObject {
    [CmdletBinding()]
    param (
        [Parameter(ValueFromPipelineByPropertyName)]
        [Alias('First', 'Second', 'Third')]
        $InputObject
    )
}
```

The alias name is considered when determining whether a property on an input object is suitable to fill a parameter when filling a parameter by property name.

One of the more common uses of this is to provide support for a `Path` parameter via a pipeline from `Get-Item` or `Get-ChildItem`. For example, the following pattern might be used to expose a `Path` parameter. This is used in the short helper

function that imports JSON content from a file:

```
function Import-Json {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory, ValueFromPipelineByPropertyName)]
        [Alias('PSPath')]
        [String]$Path
    )
    process {
        Get-Content $Path | ConvertFrom-Json
    }
}
```

The `PSPath` property of the object returned by `Get-Item` or `Get-ChildItem` is used to fill the `Path` parameter from a pipeline. `FullName` is a possible alternative to `PSPath`, depending on how the path is to be used.

Converting relative paths to full paths

When using a path parameter, such as the one in the previous example, you can use the following method on the `PSCmdlet` object to ensure that a path is fully qualified, whether it exists or not:

```
$Path = $PSCmdlet.GetUnresolvedProviderPathFromPSPath($Path)
```

This technique is useful if working with .NET types, which require a path as these are not able to resolve PowerShell paths (either relative or via a PS drive).

The `New-TimeSpan` command is an example of an existing command that uses the alias to fill a parameter from the pipeline. The `Start` parameter has an alias of `LastWriteTime`. When `Get-Item` is piped into `New-TimeSpan`, the time since the file or directory was last written will be returned as a `TimeSpan` via the aliased parameter.

Defining parameter sets

A parameter set in PowerShell groups different parameters together. In some cases, this is used to change the output of a command; in others, it provides a different way of supplying a piece of information.

For example, the output from the `Get-Process` command changes if the `Module` parameter or, to a lesser extent, the `IncludeUserName` parameter are supplied.

The `Get-ChildItem` command also has two parameter sets: one that accepts a `Path` with wildcard support and another that accepts a `LiteralPath` that does not support wildcards.

Parameter sets are declared using the `ParameterSetName` property of the `Parameter` attribute.

The following example has two parameter sets; each parameter set contains a single parameter:

```
function Get-InputObject {
    [CmdletBinding()]
    param (
        [Parameter(ParameterSetName = 'FirstSetName')]
        $Parameter1,
        [Parameter(ParameterSetName = 'SecondSetName')]
        $Parameter2
    )
}
```

As neither parameter set is the default, attempting to run the command using a positional parameter only will result in an error:

```
PS> Get-InputObject value
Get-InputObject: Parameter set cannot be resolved using the specified named parameters. One or more parameters issued cannot be used together or an insufficient number of parameters were provided.
```

This can be resolved by setting a value for the `DefaultParameterSetName` property in the `CmdletBinding` attribute:

```
[CmdletBinding(DefaultParameterSetName = 'FirstSetName')]
```

Alternatively, an explicit position might be defined for one of the parameters; the set will be selected based on explicit position:

```
[Parameter(Position = 1, ParameterSetName = 'FirstSetName')]
$Parameter1
```

The name of the parameter set within a function is visible using the `ParameterSetName` property of the `PSCmdlet` automatic variable, which is `$PSCmdlet.ParameterSetName`. You can use the value to choose actions within the body of a function.

The following example shows a possible implementation that tests the value of `ParameterSetName`. The function accepts the name of a service as a string, a service object from `Get-Service`, or a service returned from the `Win32_Service` class. The function finds the process associated with that service or fails if the service is not running:

```
function Get-ServiceProcess {
    [CmdletBinding(DefaultParameterSetName = 'ByName')]
    param (
        [Parameter(
            Mandatory,
            Position = 1,
            ParameterSetName = 'ByName'
        )]
        [String]$Name,
        [Parameter(
            Mandatory,
            ValueFromPipeline,
            ParameterSetName = 'FromService'
        )]
        [System.ServiceProcess.ServiceController]$Service,
        [Parameter(
            Mandatory,
            ValueFromPipeline,
            ParameterSetName = 'FromCimService'
        )]
       [PSTypeName('Microsoft.Management.Infrastructure.CimInstance#root/cimv2/Win32_Service')]
        [CimInstance]$CimService
    )
    process {
        if ($pscmdlet.ParameterSetName -eq 'FromService') {
            $Name = $Service.Name
        }
        if ($Name) {
            $params = @{
                ClassName = 'Win32_Service'
                Filter    = 'Name="{0}"' -f $Name
                Property  = 'Name', 'ProcessId', 'State'
            }
            $CimService = Get-CimInstance @params
        }
        if ($CimService.State -eq 'Running') {
            Get-Process -Id $CimService.ProcessId
        } else {
            Write-Error ('The service {0} is not running' -f @(
                $CimService.Name
            ))
        }
    }
}
```

The previous function accepts several different parameters. Each parameter is ultimately used to get to a value for the `$CimService` variable (or parameter), which has a `ProcessID` property associated with the service.

Each of the examples so far has shown a parameter that is a member of a single explicitly declared set.

A parameter that does not describe a `ParameterSetName` is automatically part of every set.

In the following example, `Parameter1` is part of every parameter set, while `Parameter2` is in a named set only:

```
function Test-ParameterSet {
    [CmdletBinding(DefaultParameterSetName = 'Default')]
    param (
        [Parameter(Mandatory, Position = 1)]
        $Parameter1,
        [Parameter(ParameterSetName = 'NamedSet')]
        $Parameter2
    )
}
```

`Get-Command` may be used to show the syntax for the command; this shows that there are two different parameter sets, both of which require `Parameter1`:

```
PS> Get-Command Test-ParameterSet -Syntax
Test-ParameterSet [-Parameter1] <Object> [<CommonParameters>]
Test-ParameterSet [-Parameter1] <Object> [-Parameter2 <Object>] [<CommonParameters>]
```

Parameters that do not use the `Parameter` attribute are also automatically part of all parameter sets.

A parameter may also be added to more than one parameter set. This is achieved by using more than one `Parameter` attribute:

```
function Test-ParameterSet {
    [CmdletBinding(DefaultParameterSetName = 'NamedSet1')]
    param (
        [Parameter(Mandatory)]
        $Parameter1,
        [Parameter(Mandatory, ParameterSetName = 'NamedSet2')]
        $Parameter2,
        [Parameter(Mandatory, ParameterSetName = 'NamedSet3')]
        $Parameter3,
        [Parameter(Mandatory, ParameterSetName = 'NamedSet2')]
        [Parameter(ParameterSetName = 'NamedSet3')]
        $Parameter4
    )
}
```

In the preceding example:

- `Parameter1` is in all parameter sets
- `Parameter2` is in `NamedSet2` only
- `Parameter3` is in `NamedSet3` only
- `Parameter4` is mandatory in `NamedSet2`, and optional in `NamedSet3`

This interplay of parameter sets is complex and difficult to describe without a complex command to use the parameters.

Many existing commands use complex parameter sets and their parameter sets may be explored. For example, the parameter block for the `Get-Process` command may be shown by running the following command:

```
[System.Management.Automation.ProxyCommand]::GetParamBlock((
    Get-Command Get-Process
))
```

Parameter sets are a powerful feature, allowing a command to handle different combinations of input values without resorting to complex logic in the body of a script or function.

Argument completers

The tab completion system uses argument completers to provide an argument for a parameter when Tab is pressed. For example, the `Get-Module` command cycles through module names when Tab is pressed after the command name.

Argument completers have been around in several different forms since PowerShell 2. This section focuses on the implementation of argument completers available in Windows PowerShell 5 and above.

An argument completer does not restrict the values that may be supplied for a parameter. It is used to offer values, to make it easier for an end user to figure out what to enter.

An argument completer is a script block; the script block can expect the following parameters (by position, in the order listed as follows):

- `commandName`
- `parameterName`
- `wordToComplete`
- `commandAst`
- `fakeBoundParameter`

You can use any of these parameters, but the most important and the most frequently used is `wordToComplete`.

The following example suggests words from a fixed list:

```
{
    param (
        $commandName,
        $parameterName,
        $wordToComplete,
        $commandAst,
        $fakeBoundParameter
    )
    $possibleValues = 'Start', 'Stop', 'Create', 'Delete'
    $possibleValues -like "$wordToComplete*"
}
```

Notice that a wildcard, `*`, has been added to the end of `wordToComplete`. Tab completion expects the user to have typed part of a word, but a user would not normally include a wildcard character for matching against possible values.

Arguably, `ValidateSet` might be a better option in this case as it also implicitly provides tab completion. However, where `ValidateSet` enforces, `ArgumentCompleter` suggests.

The list of possible values used in an argument completer can be dynamic. That is, the list of possible values can be the result of running another command.

PowerShell provides two different ways to assign an argument completer: the `ArgumentCompleter` attribute or the `Register-ArgumentCompleter` command.

The ArgumentCompleter attribute

The `ArgumentCompleter` attribute is used much like `ValidateScript`. You place the attribute before the parameter variable it is providing completion for.

The script block in the introduction is used as an argument for the attribute, as the following code shows:

```
function Test-ArgumentCompleter {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory)]
        [ArgumentCompleter( {
            param (
                $commandName,
                $parameterName,
                $wordToComplete,
                $commandAst,
                $fakeBoundParameter
            )
            $possibleValues = 'Start', 'Stop', 'Create', 'Delete'
            $possibleValues -like "$wordToComplete*"
        } )]
        $Action
    )
}
```

When the user types `Test-ArgumentCompleter` and then presses Tab, the completer offers up each of the possible values with

no filtering. If the user were to type `Test-ArgumentCompleters`, only `Start` and `Stop` would be offered when pressing Tab.

Using Register-ArgumentCompleter

Whereas you can use the `ArgumentCompleter` attribute to add an argument completer to a single parameter in a single command, you can use the `Register-ArgumentCompleter` command to add completion to a parameter in more than one command; this reduces code repetition.

In addition to being able to add a completer to several commands, `Register-ArgumentCompleter` can create argument completers for native commands (`.exe` files).

For example, when used as an alternative to the `ArgumentCompleter` attribute, the command is used as follows:

```
function Test-ArgumentCompleter {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory)]
        $Action
    )
}
$params = @{
    CommandName   = 'Test-ArgumentCompleter'
    ParameterName = 'Action'
    ScriptBlock   = {
        param (
            $commandName,
            $parameterName,
            $wordToComplete,
            $commandAst,
            $fakeBoundParameter
        )
        $possibleValues = 'Start', 'Stop', 'Create', 'Delete'
        $possibleValues -like "$wordToComplete*"
    }
}
Register-ArgumentCompleter @params
```

The `CommandName` parameter used for `Register-ArgumentCompleter` accepts an array of command names. In one step, the completer can be added to several different commands that share a parameter.

You can also use `Register-ArgumentCompleter` to add argument completion to native commands. The following example offers a user of the `wmic` command automatic alias completion:

```
Register-ArgumentCompleter -CommandName wmic -Native -ScriptBlock {
    param ( $wordToComplete, $commandAst, $cursorPosition )
    wmic /?:BRIEF |
        Where-Object { $_ -cmatch '^([A-Z]{2}\S+)+' } |
        ForEach-Object { $matches[1] } |
        Where-Object {
            $_ -notin 'CLASS', 'PATH', 'CONTEXT', 'QUIT/EXIT' -and
            $_ -like "$wordToComplete*"
        }
}
```

When using the `-Native` parameter, the arguments passed to the completer differ; the first argument becomes the word to complete.

Listing registered argument completers

While it is possible to register argument completers, PowerShell does not provide a way of listing them. This is somewhat

frustrating as it makes exploration and finding examples more difficult.

The following script makes extensive use of reflection in .NET to explore classes that are not made publicly available, eventually getting to a property that holds the argument completers:

```
using namespace System.Reflection
$localPipeline = [PowerShell].Assembly.GetType(
    'System.Management.Automation.Runspaces.LocalPipeline'
)
$getExecutionContextFromTLS = $localPipeline.GetMethod(
    'GetExecutionContextFromTLS',
    [BindingFlags]'Static, NonPublic'
)
$internalExecutionContext = $getExecutionContextFromTLS.Invoke(
    $null,
    [BindingFlags]'Static, NonPublic',
    $null,
    $null,
    $psculture
)
$customArgumentCompletersProperty = $internalExecutionContext.GetType().GetProperty(
    'CustomArgumentCompleters',
    [BindingFlags]'NonPublic, Instance'
)
$customArgumentCompletersProperty.GetGetMethod($true).Invoke(
    $internalExecutionContext,
    [BindingFlags]'Instance, NonPublic, GetProperty',
    $null,
    @(),
    $psculture
)
```

Native argument completers are held in a different property and will not be shown by the previous snippet.

> A more refined version of the previous snippet, which also supports the retrieval of native argument completers, is available as a function at:
>
> https://gist.github.com/indented-automation/26c637fb530c4b168e62c72582534f5b

Argument completers are a fantastic way to suggest values for parameters without constraining the possible input values.

So far in this chapter, all the parameters have been explicitly defined in a `param` block, and any attributes have been explicitly added. In PowerShell, it is possible to define parameters dynamically.

Dynamic parameters

Dynamic parameters allow a developer to define the behavior of parameters when a function or script is run, rather than hardcoding that behavior in advance in a `param` block.

Dynamic parameters can be used to overcome some of the limitations inherent in a `param` block. For example, it is possible to change the parameters presented by a command based on the value of another parameter. It is also possible to dynamically write validation, such as dynamically assigning a value for the `ValidateSet` attribute.

Dynamic parameters remain unpopular in the PowerShell community. They are relatively complex; that is, they are hard to define and difficult to troubleshoot as they tend to silently fail rather than raising an error.

Dynamic parameters have a named block: `dynamicparam`. If you use `dynamicparam` in a script or function, the implicit default blocks for a script or function cannot be used; all code must be contained within an explicitly named block.

The `CmdletBinding` attribute must be explicitly declared when using `dynamicparam`; you cannot create parameters without

`CmdletBinding`, nor will an error message be shown to explain that.

The following example includes an empty `dynamicparam` block as well as an `end` block, which would have been implicit if `dynamicparam` were not present:

```
function Test-DynamicParam {
    [CmdletBinding()]
    param ( )
    dynamicparam { }
    end {
        Write-Host 'Function body'
    }
}
```

If the `end` block declaration is missing, a syntax error will be displayed.

The following example will cause an error if pasted into the console or if the code is executed via an editor:

```
function Test-DynamicParam {
    [CmdletBinding()]
    param ( )
    dynamicparam { }
    Write-Host 'Function body'
}
```

In PowerShell 7, the first error message will explicitly state that `Write-Host` is unexpected and that a named block is expected:

```
ParserError:
Line |
   7 |      Write-Host 'Function body'
     |      ~~~~~~~~~~
     | unexpected token 'Write-Host', expected 'begin', 'process', 'end', or 'dynamicparam'.
```

Windows PowerShell is less clear; it states that a closing brace is missing:

```
At line:1 char:28
+ function Test-DynamicParam {
+                            ~
Missing closing '}' in statement block or type definition.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : MissingEndCurlyBrace
```

The `dynamicparam` block must create and return a `RuntimeDefinedParameterDictionary` object. The dictionary can contain zero or more `RuntimeDefinedParameter` objects.

Creating a RuntimeDefinedParameter object

A `RuntimeDefinedParameter` object describes a single parameter. The definition includes the name of the parameter, the parameter type, and any attributes that should be set for that parameter. PowerShell does not include type accelerators for creating `RuntimeDefinedParameter`; the full name, `System.Management.Automation.RuntimeDefinedParameter`, must be used.

The constructor for `RuntimeDefinedParameter` expects three arguments: a string, which is the parameter name, a .NET type for the parameter, and a collection or array of attributes that should be assigned. The attribute collection must contain at least one `Parameter` attribute.

The following example, which creates a parameter named `Action`, makes use of a `using namespace` statement to shorten the .NET type names:

```
using namespace System.Management.Automation
$parameter = [RuntimeDefinedParameter]::new(
    'Action',
    [String],
    [Attribute[]]@(
        [Parameter]@{ Mandatory = $true; Position = 1 }
        [ValidateSet]::new('Start', 'Stop', 'Create', 'Delete')
    )
)
```

The previous parameter is the equivalent of using the following in the `param` block:

```
param (
    [Parameter(Mandatory, Position = 1)]
    [ValidateSet('Start', 'Stop', 'Create', 'Delete')]
    [String]$Action
)
```

As the attributes are not being placed directly above a variable, each must be created as an independent object instance:

- The shorthand used for the `Parameter` attribute in the `param` block cannot be used; expressions cannot be omitted when describing properties.
- The `ValidateSet` attribute, and other validation attributes, must also be created as a new object rather than using the attribute syntax.

The `Parameter` attribute declaration takes advantage of PowerShell's ability to assign property values to an object using a Hashtable. This is feasible because a `Parameter` attribute can be created without supplying any arguments, that is, `[Parameter]::new()` can be used to create a `Parameter` attribute with default values. This technique cannot be used for the validation attributes, as each requires one or more arguments, therefore, you must use `::new` or `New-Object`.

As with a normal parameter, `RuntimeDefinedParameter` can declare more than one parameter attribute. Each `Parameter` attribute is added to the attribute collection:

```
using namespace System.Management.Automation
$parameter = [RuntimeDefinedParameter]::new(
    'Action',
    [String],
    [Attribute[]]@(
        [Parameter]@{
            Mandatory        = $true
            Position         = 1
            ParameterSetName = 'First'
        }
        [Parameter]@{
            ParameterSetName = 'Second'
        }
    )
)
```

You can create any number of parameters in this manner. Each parameter must have a unique name. Each parameter must be added to a `RuntimeDefinedParameterDictionary`.

Using RuntimeDefinedParameterDictionary

`RuntimeDefinedParameterDictionary` is the expected output from the `dynamicparam` block. The dictionary must contain all the dynamic parameters a function is expected to present.

The following example creates a dictionary and adds a single parameter:

```
using namespace System.Management.Automation
function Test-DynamicParam {
    [CmdletBinding()]
    param ( )
    dynamicparam {
        $parameters = [RuntimeDefinedParameterDictionary]::new()
        $parameter = [RuntimeDefinedParameter]::new(
            'Action',
            [String],
            [Attribute[]]@(
                [Parameter]@{ Mandatory = $true; Position = 1 }
                [ValidateSet]::new(
                    'Start',
                    'Stop',
                    'Create',
                    'Delete'
                )
            )
        )
        $parameters.Add($parameter.Name, $parameter)
        $parameters
    }
}
```

Using dynamic parameters

Dynamic parameters must be accessed using the `PSBoundParameters` variable within a function or script; dynamic parameters do not initialize variables of their own.

The value of the `Action` parameter used in the previous examples must be retrieved by using `Action` as a key, shown as follows:

```
using namespace System.Management.Automation
function Test-DynamicParam {
    [CmdletBinding()]
    param ( )
    dynamicparam {
        $parameters = [RuntimeDefinedParameterDictionary]::new()
        $parameter = [RuntimeDefinedParameter]::new(
            'Action',
            [String],
            [Attribute[]]@(
                [Parameter]@{ Mandatory = $true; Position = 1 }
                [ValidateSet]::new(
                    'Start',
                    'Stop',
                    'Create',
                    'Delete'
                )
            )
        )
        $parameters.Add($parameter.Name, $parameter)
        $parameters
    }
    end {
        Write-Host $PSBoundParameters['Action']
    }
}
```

As with parameters from the `param` block, the `$PSBoundParameters.ContainsKey` method may be used to find out whether a user has specified a value for the parameter. Dynamic parameters cannot have a default value; any default values must be

created in `begin`, `process`, or `end`.

A dynamic parameter that accepts pipeline input, like a normal parameter that accepts pipeline input, will only have a value within the `process` and `end` blocks. The `end` block will only see the last value in the pipeline. The following example demonstrates this:

```
using namespace System.Management.Automation
function Test-DynamicParam {
    [CmdletBinding()]
    param ( )
    dynamicparam {
        $parameters = [RuntimeDefinedParameterDictionary]::new()
        $parameter = [RuntimeDefinedParameter]::new(
            'InputObject',
            [String],
            [Attribute[]]@(
                [Parameter]@{
                    Mandatory         = $true
                    ValueFromPipeline = $true
                }
            )
        )
        $parameters.Add($parameter.Name, $parameter)
        $parameters
    }
    begin {
        'BEGIN: Input object is present: {0}' -f @(
            $PSBoundParameters.ContainsKey('InputObject')
        )
    }
    process {
        'PROCESS: Input object is present: {0}; Value: {1}' -f @(
            $PSBoundParameters.ContainsKey('InputObject')
            $PSBoundParameters['InputObject']
        )
    }
    end {
        'END: Input object is present: {0}; Value: {1}' -f @(
            $psboundparameters.ContainsKey('InputObject')
            $psboundparameters['InputObject']
        )
    }
}
```

The function can be used with arbitrary input values, for example:

```
PS> 1..2 | Test-DynamicParam
BEGIN: Input object is present: False
PROCESS: Input object is present: True; Value: 1
PROCESS: Input object is present: True; Value: 2
END: Input object is present: True; Value: 2
```

The `$PSBoundParameters` variable, and any other parameters, may be referenced inside the `dynamicparam` block.

Conditional parameters

One possible use of dynamic parameters is to change validation based on the value supplied for another parameter. Another use is to change which parameters are available, again based on the value of another parameter.

The following example changes `validValues` into `ValidateSet` depending on the value supplied for the `Type` parameter:

```
using namespace System.Management.Automation
function Test-DynamicParam {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory, Position = 1)]
        [ValidateSet('Service', 'Process')]
        [String]$Type,
        [Parameter(Mandatory, Position = 3)]
        [String]$Name
    )
    dynamicparam {
        $parameters = [RuntimeDefinedParameterDictionary]::new()
        [String[]]$validValues = switch ($Type) {
            'Service' { 'Get', 'Start', 'Stop', 'Restart' }
            'Process' { 'Get', 'Kill' }
        }
        $parameter = [RuntimeDefinedParameter]::new(
            'Action',
            [String],
            [Attribute[]]@(
                [Parameter]@{ Mandatory = $true; Position = 2 }
                [ValidateSet]::new($validValues)
            )
        )
        $parameters.Add($parameter.Name, $parameter)
        $parameters
    }
}
```

Changing validation in this manner is entirely reliant on the user having typed a value for the `Type` parameter before attempting to type `Action`. You can make other comparisons in dynamic parameter blocks; for example, a parameter might only appear when a certain condition is met. `RuntimeDefinedParameterDictionary` is valid even if it is empty and no extra parameters need to be added.

Summary

You can set attributes on parameters in PowerShell to quickly and easily define the behavior, acceptable values, and usage for a parameter. These attributes greatly simplify the validation that may be required in the body of a script or function.

PowerShell comes with a wide range of built-in validators, and each of the existing validators is briefly demonstrated in this chapter. As well as validation, extra controls can be placed around the content of a parameter, such as whether empty strings or collections may be used as a value.

The `Parameter` attribute is incredibly important in PowerShell as it allows pipeline input support to be declared for a parameter, and for parameters to be placed in different parameter sets.

Argument completers have changed a great deal as PowerShell has progressed through each version. In PowerShell 5 and above, you can use the `ArgumentCompleter` attribute and the `Register-ArgumentCompleter` command to add tab completion support to a parameter.

Dynamic parameters are an interesting feature of PowerShell, although a feature that should be used with care as it is easy to break. The examples in this chapter presented the creation of dynamic parameters using the simplest syntax possible.

The next chapter, *Classes and Enumerations*, explores the features available to classes that were introduced with PowerShell 5.

19

Classes and Enumerations

PowerShell 5 introduced support for creating classes and enumerations within PowerShell directly. Prior to this, you had to import classes from an assembly written in a language such as C# or create them dynamically, which is complex and beyond the scope of this chapter.

Classes and enumerations have only changed a little in PowerShell 6 and 7. There are numerous enhancement issues open in the PowerShell project on GitHub, but these are slow to make their way into PowerShell. Examples include support for writing interfaces and the ability to define getters and setters for properties.

This chapter explores the following topics:

- Defining an enumeration
- Creating a class
- Argument transformation attribute classes
- Validation attribute classes
- Classes and DSC

Defining an enumeration

An enumeration is a set of named constants. .NET is full of examples of enumerations. For example, the `System.Security.AccessControl.FileSystemRights` enumeration describes all of the numeric values that are used to define access rights for files or directories.

Enumerations are also used by PowerShell itself. For example, `System.Management.Automation.ActionPreference` contains the values for the preference variables, such as `ErrorActionPreference` and `DebugPreference`.

You create enumerations by using the `enum` keyword followed by a list of names and values:

```
enum MyEnum {
    First  = 1
    Second = 2
    Third  = 3
}
```

Each name must be unique within the enumeration and must start with a letter or an underscore. The name may contain numbers after the first character. The name cannot be quoted and cannot contain the hyphen character.

The value does not have to be unique. One or more names in an enumeration can share a single value:

```
enum MyEnum {
    One    = 1
    First  = 1
    Two    = 2
    Second = 2
}
```

By default, enumeration values are `Int32`, which is the underlying type.

Enum and underlying types

In languages such as C#, enumerations can be given an underlying type, such as `Byte` or `Int64`. In PowerShell 5 and PowerShell Core 6.1 and older, the enumeration type is fixed to `Int32`. This type is shown using the following command:

```
PS> enum MyEnum {
>>     First = 1
>> }
PS> [MyEnum].GetEnumUnderlyingType()
IsPublic    IsSerial    Name      BaseType
--------    --------    ----      --------
True        True        Int32     System.ValueType
```

In PowerShell 7, you may specify the underlying type for an enumeration:

```
enum MyEnum : UInt64 {
    First = 0
    Last  = 18446744073709551615
}
```

You can use any integer type as the value type, including `byte`, `short` (`Int16`), `uint`, and (`UInt32`).

An enumeration value may be cast to its underlying type to reveal the number behind the name:

```
[Int][MyEnum]::First
```

Alternatively, you can use the `-as` operator:

```
[MyEnum]::First -as [MyEnum].GetEnumUnderlyingType()
```

The numeric value is also accessible using the `value__` property:

```
[MyEnum]::First.value__
```

The preceding example enumerations have explicit values defined for each name. Enumerations support automatic numbering as the next section shows.

Automatic value assignment

An enumeration may be created without defining a value for a name. PowerShell will automatically allocate a sequence of values starting from `0`. In the following example, the names `Zero` and `One` are automatically created with the values `0` and `1`, respectively:

```
enum MyEnum {
    Zero
    One
}
```

If a value is assigned to a name, the sequence will continue from that point. The following example starts with the value `5`. The name `Six` will automatically be given the value `6`:

```
enum MyEnum {
    Five = 5
    Six
}
```

Automatic value assignment supports non-contiguous values. The sequence may be restarted at any point, or values may be skipped. The following example demonstrates both restarting a sequence and skipping values in a sequence:

```
enum MyEnum {
    One   = 1
    Two
    Five  = 5
    Six
    First = 1
    Second
}
```

The previous example mixes two potentially different name sets in a single enumeration to demonstrate restarting the numeric sequence. It is rarely desirable to mix names and values in this manner. If a distinct set of constant names is needed for a set of names, a second enumeration is a better approach.

You can use enumerations to restrict arguments for parameters in a similar manner to the `ValidateSet` attribute.

Enum or ValidateSet

It is common to create functions or scripts with parameters that accept a fixed set of values. `ValidateSet` can be used to limit the possible arguments for a parameter.

The following example uses the values `Absent` and `Present`, for a parameter named `Ensure`. This parameter and these values are commonly used when writing **Desired State Configuration** (**DSC**) resources, which are explored further at the end of this chapter:

```
function Test-ParameterValue {
    param (
        [Parameter(Mandatory)]
        [ValidateSet('Absent', 'Present')]
        [string]$Ensure
    )
}
```

The `ValidateSet` attribute used in the preceding example restricts the parameter to one of the two different values. As many values as required can be used with `ValidateSet`.

Alternatively, you can use an enumeration to restrict values. This has the benefit that the same set of values can be shared across several different functions in a script or module.

The following enumeration describes the same possible values used in `ValidateSet` above:

```
enum Ensure {
    Absent
    Present
}
function Test-ParameterValue {
    param (
        [Parameter(Mandatory)]
        [Ensure]$Ensure
    )
}
```

In some cases, the numeric value may be irrelevant, which can be true of the values for preference variables in PowerShell for instance. The value for a preference variable can be numeric but it is far more common to use a string. For example, the following two preference variable assignments are equivalent:

```
$VerbosePreference = 'Continue'
$VerbosePreference = 2
```

The origin of the numeric value used above is shown in the following line of code:

```
[System.Management.Automation.ActionPreference]::Continue.value__
```

The numeric value may be considered important again when it has a parallel with another value type. In the case of the `Ensure` enumeration it is most logical to have `Absent` represented by `0`, and `Present` by `1`.

The enumerations used previously are used as single values. The name used with the parameter is either `Absent` or `Present`, never a combination of the two. Enumerations that support more than one name being used are created using the `Flags` attribute.

The Flags attribute

An enumeration that uses the `Flags` attribute allows you to use more than one name when describing a value. In *Chapter 10, Files, Folders, and the Registry*, the `System.Security.AccessControl.FileSystemRights` enumeration was used to describe access rights. Access rights are described as a combination of several enumeration values. The names are written as a comma-separated list. For example:

```
[System.Security.AccessControl.FileSystemRights]'ReadData, Delete'
```

This is only possible because the enumeration has the `Flags` attribute set:

```
PS> $enumType = [System.Security.AccessControl.FileSystemRights]
PS> $enumType.CustomAttributes.AttributeType.Name
FlagsAttribute
```

The `Flags` attribute can be used when creating enumerations in PowerShell. The attribute is placed before the `enum` keyword as shown here:

```
[Flags()]
enum MyEnum {
    First  = 1
    Second = 2
    Third  = 4
}
```

Each numeric value in the enumeration has a distinct combination of (binary) bits set to make the value unique:

```
[Flags()]
enum MyEnum {
    First  = 1 # 001
    Second = 2 # 010
    Third  = 4 # 100
}
```

The value of the flag will therefore double each time: `1`, `2`, `4`, `8`, `16`, `32`, `64`, `128`, and so on. Enumeration values, especially flags, are frequently written in hexadecimal:

```
[Flags()]
enum MyEnum {
    First  = 0x001  # 1
    Second = 0x002  # 2
    Third  = 0x004  # 4
    Fourth = 0x008  # 8
    Fifth  = 0x010  # 16
    Sixth  = 0x020  # 32
}
```

In many respects it is easier to read the value in hexadecimal. The unique values are always `1`, `2`, `4`, and `8`, with a shift in

position each time the value appears: `0x001` the first time, `0x010` the second, `0x100` the third, and so on.

As each number doubles, automatic value assignment does not create useful values (beyond the `1` and `2`) if the `Flags` attribute is used.

PowerShell will cast a numeric value into a set of names if it can be matched in a `Flags` enumeration. A value of `6` can be used to represent the `Second` and `Third` flags for example:

```
PS> [MyEnum]6
Second, Third
```

Several enumerations that use the `Flags` attribute also provide named composite values. For example, `System.Security.AccessControl.FileSystemRights` defines several permissions as combinations of other values. The `ReadAndExecute` right is made up of the individual flags `ReadData`, `ReadExendedAttributes`, and `ReadPermissions`. You can use the following function to reveal these individual flag values from a combined or composite value:

```
function Get-FlagName {
    [CmdletBinding()]
    param (
        $Value
    )
    $enumType = $Value.GetType()
    $i = 0
    do {
        $bit = $Value -as [Int64] -band 1 -shl $i++
        if ($bit) {
            [PSCustomObject]@{
                Name        = $bit -as $enumType
                Integer     = $bit
                Hexadecimal = '0x{0:X8}' -f $bit
                BitPosition = $i
            }
        }
    } until (1 -shl $i -ge $value)
}
```

The function works by taking the value `1` then bit-shifting it one place to the left with each iteration of the loop. That is, it compares the value `1`, then `2`, `4`, `8`, `16`, and so on. If that exact bit in the composite value is set, the custom object describing that value is returned.

Running the function with the `ReadAndExecute` enumeration value shows the individual flags that make up the composite value:

```
PS> using namespace System.Security.AccessControl
PS> Get-FlagName -Value ([FileSystemRights]::ReadAndExecute)
                   Name Integer Hexadecimal BitPosition
                   ---- ------- ----------- -----------
               ReadData       1 0x00000001           1
ReadExtendedAttributes       8 0x00000008           4
            ExecuteFile      32 0x00000020           6
         ReadAttributes     128 0x00000080           8
        ReadPermissions  131072 0x00020000          18
```

You can create composite values in a PowerShell enumeration by setting a value that has more than one flag set. For example, the tweak to the example enumeration adds a composite value for the `First` and `Third` flags:

```
[Flags()]
enum MyEnum {
    First        = 1 # 001
    Second       = 2 # 010
    Third        = 4 # 100
    FirstAndThird = 5 # 101
}
```

As `FirstAndThird` explicitly matches the value `5`, any value the enumeration converts will use the `FirstAndThird` name instead of the individual values:

```
PS> [MyEnum]7
Second, FirstAndThird
PS> [MyEnum]'First, Second, Third'
Second, FirstAndThird
```

You can use the preceding function on the tweaked enumeration to show the individual values:

```
PS> Get-FlagName -Value ([MyEnum]::FirstAndThird)
Name Integer Hexadecimal BitPosition
 ---- ------- ----------- -----------
First       1 0x00000001           1
Third       4 0x00000004           3
```

As enumerations associate a name with a number, you can use enumerations to convert different related names.

Using enumerations to convert a value

Enumerations are lists of names, each assigned a numeric value. You can use a pair of enumerations to convert between two lists of names, linked by a common number.

The following example defines two enumerations. The first is a list of values the end user will see; the second holds the internal name required by the code. This simulates, in part, the type of aliasing performed by the `wmic` command:

```
enum AliasName {
    OS
    Process
}
enum ClassName {
    Win32_OperatingSystem
    Win32_Process
}
```

A function can use the `AliasName` enumeration, as shown here:

```
function Get-CimAliasInstance {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory, Position = 1)]
        [AliasName]$AliasName
    )
    Get-CimInstance -ClassName ([ClassName]$AliasName)
}
```

The command may now be used with the `OS` argument for the `AliasName` parameter as demonstrated in the following trimmed output:

```
PS> Get-CimAliasInstance -AliasName OS
SystemDirectory     Organization BuildNumber
---------------     ------------ -----------
C:\WINDOWS\system32              19042
```

This will be converted to `Win32_OperatingSystem` by way of the enumeration. `Get-CimInstance` handles converting that value into a string.

Creating a class

A class is used to describe an object. This may be any object, which means that a class in PowerShell might be used for any purpose.

Classes in PowerShell are created using the `class` keyword. The following class contains a single property:

```
class MyClass {
    [string]$Value = 'My value'
}
```

You can create the class by using either the `New-Object` command or the `::new()` method:

```
PS> New-Object MyClass
Value
-----
My value
PS> [MyClass]::new()
Value
-----
My Value
```

This example creates an instance of the class and displays the property that was defined for the class.

Properties

The properties defined in a class may define a .NET type and may have a default value if required. The following class has a single property with the `String` type:

```
class MyClass {
    [string]$Value = 'My value'
}
```

PowerShell automatically adds hidden `get` and `set` methods used to access the property; these cannot be overridden or changed at this time (within the class itself). The `get` and `set` methods may be viewed using `Get-Member` with the `Force` parameter:

```
PS> [MyClass]::new() | Get-Member get_*, set_* -Force
   TypeName: MyClass
Name         MemberType    Definition
----         ----------    ----------
get_Value    Method        string get_Value()
set_Value    Method        void set_Value(string )
```

You can access the property, and any other properties, on an instance of the class:

```
PS> $instance = [MyClass]::new()
PS> $instance.Value
My value
```

Classes are created with a default constructor that requires no arguments unless a constructor is explicitly defined.

Constructors

A constructor is executed when either `New-Object` or `::new()` is used to create an instance of a class. The implicit or default constructor does not require arguments.

An explicit constructor may be created to handle more complex actions when creating an object.

The constructor has the same name as the class. The reserved variable `$this` is used to refer to properties and methods within the class. The following constructor sets a value for `Value` when an instance of the class is created:

```
class MyClass {
    [string]$Value
    MyClass() {
        $this.Value = 'Hello world'
    }
}
```

The constructor may require arguments to create an instance of the class. The following example requires a string, updates the value, then sets it as the value of the `Value` property:

```
class MyClass {
    [string]$Value
    MyClass(
        [string] $Argument
    ) {
        $culture = Get-Culture
        $this.Value = $culture.TextInfo.ToTitleCase(
            $Argument
        )
    }
}
```

The argument for the constructor is passed when creating the instance of the object. The result is:

```
PS> [MyClass]::new('hello world')
Value
-----
Hello World
```

The preceding `ToTitleCase` method has capitalized the first characters of each word in the string before assigning it to the `Value` property.

So far it can be said that:

- Default values may be assigned to properties as required
- If work is required to fill a property (or properties) when an object is created, a constructor with no arguments can be used
- If work is required to fill a property (or properties), and an argument should be supplied by the consumer (end user, or another piece of code) of the class, a constructor can be created that accepts arguments

Constructors may be overloaded to allow the class to accept different arguments or combinations of arguments when the class is created.

The following example has two constructors, first a default constructor that sets a default value for the `Value` property. The

second constructor allows the consumer of the class to define a value:

```
class MyClass {
    [string]$Value
    MyClass() {
        $this.Value = 'Hello world'
    }
    MyClass(
        [string] $Argument
    ) {
        $culture = Get-Culture
        $this.Value = $culture.TextInfo.ToTitleCase(
            $Argument
        )
    }
}
```

Each constructor must have a unique signature. The signature for the constructor is based on the number of arguments, and the types (such as `String` as used above) of those arguments. Each combination must be unique.

If you need to enclose shared functionality or want to allow a consumer to enact a change in a class, then you'll need to create a method.

Methods

A method causes a change to the object. This may be an internal change, such as opening a connection or stream, or it may take the object and change it into a different form as is the case with the `ToString` method.

With the class above, if `ToString` is called the name of the class is returned.

```
PS> class MyClass {
>>     [string]$Value = 'Hello world'
>> }
PS> [MyClass]::new().ToString()
MyClass
```

This is the default implementation for the `ToString` method. The method can be replaced in the class so that it returns the value of the property:

```
class MyClass {
    [string]$Value = 'Hello world'
    [string] ToString() {
        return $this.Value
    }
}
```

Running the `ToString` method this time shows the value of the `Value` property:

```
PS> [MyClass]::new().ToString()
Hello world
```

When working with methods, and unlike functions in PowerShell, the `return` keyword is mandatory. Methods do not return output by default. An error will be raised if a method has an output type declared and it does not return output from each code path.

Methods can accept arguments in the same way as constructors. Methods can also be overloaded. For example, the `ToString` method might be overloaded, providing support for output formatting. An example of this is shown here:

```
class MyClass {
    [string]$Value = 'Hello world'
    [string] ToString() {
        return '{0} on {1}' -f  @(
            $this.Property
            (Get-Date).ToShortDateString()
        )
    }
    [string] ToString(
        $dateFormat
    ) {
        return '{0} on {1}' -f @(
            $this.Property
            Get-Date -Format $format
        )
    }
}
```

The arguments supplied will dictate which method implementation is used.

Properties and methods in a class can be hidden from casual viewing using the `Hidden` modifier.

The Hidden modifier

You can use the `Hidden` modifier to hide a property from tab completion and `Get-Member`.

Members marked as `Hidden` can still be seen if the `-Force` parameter is used with `Get-Member` and `Hidden` members may still be accessed or invoked. In many respects, `Hidden` is similar to the `DontShow` property of the `Parameter` attribute.

In the following example, the `Initialize` method is hidden:

```
class MyClass {
    [string]$Property
    MyClass() {
        $this.Initialize()
    }
    hidden [void] Initialize() {
        $this.Property = 'defaultValue'
    }
}
```

You can use `Get-Member` to show this method:

```
PS> [MyClass]::new() | Get-Member Initialize -Force
    TypeName: MyClass
Name          MemberType    Definition
----          ----------    ----------
Initialize    Method        void Initialize()
```

The properties and methods in the classes demonstrated so far require an instance of the type to be created before they can be used.

The Static modifier

All of the members demonstrated so far have required the creation of an instance of a type using either `New-Object` or `::new()`.

You can execute static members without creating an instance of a type (based on a class).

Classes may implement static properties and static methods using the `Static` modifier keyword:

```
class MyClass {
    static [string] $Property = 'Property value'
    static [string] Method() {
        return 'Method return'
    }
}
```

The static members are invoked as follows:

```
[MyClass]::Property
[MyClass]::Method()
```

The `Hidden` modifier may be used in conjunction with the `Static` modifier. The modifiers may be used in either order.

Classes can inherit properties and methods from other classes.

Inheritance

Classes in PowerShell can inherit from other classes, from classes in PowerShell and .NET. The properties and methods in a base class are available to an inheriting class.

The following example defines two classes – the second inherits from the first:

```
class MyBaseClass {
    [string]$BaseProperty = 'baseValue'
}
class MyClass : MyBaseClass {
    [string]$Property = 'Value'
}
```

The `BaseProperty` property is made available on instances of the child class:

```
PS> [MyClass]::new()
Property    BaseProperty
--------    ------------
Value       baseValue
```

Members may be overridden by redeclaring the member on the inheriting class. The `GetString` method implementation from the base class is overridden in the following example:

```
class MyBaseClass {
    [string] GetString() { return 'default' }
}
class MyClass : MyBaseClass {
    [string] GetString() { return 'new' }
}
```

Each class can inherit from one other class only, but there is no limit to how deep inheritance can go. However, the depth should be limited to a sensible degree, perhaps no more than 5 or so classes deep.

Constructors are also accessible via inherited classes, but each constructor must be defined in a child class. Constructor inheritance is not entirely automatic.

Constructors and inheritance

Constructors in an inheritance hierarchy are not overridden, but instead executed in sequence.

In the following example, both classes have a default constructor (a constructor that does not require any arguments):

```
class ParentClass {
    ParentClass() {
        Write-Host 'Default parent constructor'
    }
}
class ChildClass : ParentClass {
    ChildClass() {
        Write-Host 'Default child constructor'
    }
}
```

When an instance of `ChildClass` is created, the constructor on the parent class executes first, followed by the constructor on the child class:

```
PS> $instance = [ChildClass]::new()
Default parent constructor
Default child constructor
```

If an overload is added to the constructor on `ChildClass`, PowerShell will run the new constructor, and the default constructor on `ParentClass`. The changed classes are as follows:

```
class ParentClass {
    ParentClass() {
        Write-Host 'Default parent constructor'
    }
}
class ChildClass : ParentClass {
    ChildClass() {
        Write-Host 'Default child constructor'
    }
    ChildClass([string]$value) {
        Write-Host 'Overloaded child constructor'
    }
}
```

Creating an instance of `ChildClass` shows that both constructors are called:

```
PS> $instance = [ChildClass]::new('value')
Parent constructor
Overloaded child constructor
```

Automatic execution of the default constructor in an inherited class works provided the inherited class has a default constructor. In the following example, the default constructors are removed from both classes:

```
class ParentClass {
    ParentClass([string]$value) {
        Write-Host 'Overloaded parent constructor'
    }
}
class ChildClass : ParentClass {
    ChildClass([string]$value) {
        Write-Host 'Overloaded child constructor'
    }
}
```

Attempting to create a new instance of `ChildClass` will now result in an error. PowerShell expects to be able to invoke the default constructor in `ParentClass` and that no longer exists.

```
PS> $instance = [ChildClass]::new('value')
MethodException:
Line |
   2 |      ChildClass([string]$value) {
     |                                  ~
     | Cannot find an overload for "new" and the argument count: "0".
```

Without the default constructor, `ChildClass` must be told how it is to build an instance of `ParentClass`. The `base` keyword is used to achieve this:

```
class ParentClass {
    ParentClass([string]$value) {
        Write-Host 'Non-default parent constructor'
    }
}
class ChildClass : ParentClass {
    ChildClass([string]$value) : base($value) {
        Write-Host 'Non-default child constructor'
    }
}
```

The `base` keyword can only be used with constructors, not methods.

Creating an instance of `ChildClass` this time shows both constructors run again:

```
PS> $instance = [ChildClass]::new('value')
Non-default parent constructor
Non-default child constructor
```

In the previous example, the variable `$value` is accepted by the constructor on the `ChildClass`, and explicitly passed as an argument to the constructor in `ParentClass`.

Extending on the preceding example, the constructor in the `ChildClass` can be changed again, this time removing the argument. The `base` keyword is used with a string to invoke the constructor in `ParentClass`.

```
class ParentClass {
    ParentClass([string]$value) {
        Write-Host 'Non-default parent constructor'
    }
}
class ChildClass : ParentClass {
    ChildClass() : base('Any string value') {
        Write-Host 'Default child constructor'
    }
}
```

Once again, creating an instance of `ChildClass` shows which constructors are being used:

```
PS> $instance = [ChildClass]::new()
Non-default parent constructor
Default child constructor
```

The constructor in the parent class is matched by the names and types of the arguments for the `base` keyword.

Using this, it can be said that:

- By default, all constructors in a child class call the default constructor in a parent class
- If a default constructor does not exist in a parent class, the `base` keyword must be used with every constructor in a child class to specify a constructor in the parent class

The `base` keyword cannot be applied to methods in classes. A different technique must be used when invoking methods of the same name in a parent class.

Calling methods in a parent class

Methods declared in a child class override the method defined in a parent class. Methods in a parent class are not implicitly called.

It is possible to invoke methods in a child class by casting to the parent class type. This is useful if, for example, the method in the child class adds more specialized functionality than the original method.

```
class ParentClass {
    [string] GetString() {
        return 'Hello world'
    }
}
class ChildClass : ParentClass {
    [string] GetString() {
        $string = ([ParentClass]$this).GetString()
        return '{0} on {1}' -f @(
            $string
            Get-Date -Format 'dddd'
        )
    }
}
```

The method is invoked on an instance of `ChildClass`:

```
PS> [ChildClass]::new().GetString()
Hello world on Sunday
```

If the cast to `ParentClass` were omitted, the method would start an infinite loop.

The inheritance notation can be used to implement an interface.

Working with interfaces

An interface is a set of instructions for implementing a specific behavior in a class. .NET includes many different interfaces; some of the more common examples include `IComparable` and `IEquatable`.

Implementing `IComparable` defines how an instance of a class should be compared using the `-gt`, `-ge`, `-lt`, and `-le` operators. `IComparable` also provides support for sorting.

Implementing `IEquatable` defines how an instance of a class should be compared using the `-eq` and `-ne` operators. While at first glance, the two might seem to overlap, the sections that follow show this is not the case.

Declaring support for an interface uses the same notation as inheritance; however, unlike class inheritance a class can implement more than one interface.

To demonstrate these interfaces, a simple class with a single property that has a numeric value is used:

```
class MyClass {
    [int] $Number
}
```

The numeric value will be used as the basis for the comparisons used below. The choice of value (and value type) is otherwise arbitrary.

Each interface defines the methods a class must implement to support that interface.

Implementing IComparable

As mentioned above, the `IComparable` interface makes it possible to usefully compare two instances of a class using `-gt`, `-ge`, `-lt`, and `-le`. It also makes it possible to sort instances of a class without naming a property to sort on.

`IComparable` must be implemented as described on Microsoft Docs: https://docs.microsoft.com/dotnet/api/system.icomparable.

Classes implementing `IComparable` must therefore define a `CompareTo` method that returns a single integer value, `1`, `-1`, or `0`.

Microsoft Docs includes a second version of `IComparable`, `IComparable<T>`. Limitations in the class implementation in PowerShell make the two indistinguishable so the simpler version is used.

Support for `IComparable` in the class is therefore added as shown here:

```
class MyClass : IComparable {
    [int] $Number
    [int] CompareTo([object] $object) {
        if ($this.Number -gt $object.Number) {
            return 1
        } elseif ($this.Number -lt $object.Number) {
            return -1
        } else {
            return 0
        }
    }
}
```

Once the method and the interface declaration exist, one instance of the class can be compared to another:

```
PS> $first = [MyClass]@{ Number = 1 }
PS> $second = [MyClass]@{ Number = 2 }
PS> $first -lt $second
True
PS> $first -gt $second
False
```

If the value of the `Number` property in both instances of the class is the same, then `-ge` and `-le` will return `true`:

```
PS> $first = [MyClass]@{ Number = 1 }
PS> $second = [MyClass]@{ Number = 1 }
PS> $first -ge $second
True
PS> $first -le $second
True
```

However, because `IComparable` does not specifically allow the testing of equality, using the `-eq` operator will return `false`:

```
PS> $first -eq $second
False
```

The `IEquatable` interface is used to support equality comparisons.

Implementing IEquatable

You can use the `IEquatable` interface to support comparisons using `-eq` and `-ne`.

Microsoft Docs shows `IEquatable` with a type argument, indicated by the `<T>` in the type name: https://docs.microsoft.com/dotnet/api/system.iequatable-1.

The documentation shows that any class implementing the `IEquatable` interface must implement an `Equals` method that returns a Boolean value and a `GetHashCode` method.

The type used is expected to be the same type as the class `IEquatable` is being implemented for. That is, an implementation of `IEquatable` would normally expect to start as follows:

```
class MyClass : IEquatable[MyClass] {
```

PowerShell is not able to support this. The following error will be shown if the preceding approach is used:

```
ParentContainsErrorRecordException: An error occurred while creating the pipeline.
```

To allow PowerShell to create the class, we'll use `[object]` as the type in place of the more specific `MyClass`. This is shown in the following example, which also implements `GetHashCode`.

`GetHashCode` should provide a unique value describing the object. As the basis for all comparisons with this object is the value of the `Number` property, the value of the property is also used as the hash code. The number 1, after all, uniquely identifies the number 1 when compared with all other numbers.

`GetHashCode` is a complex and involved topic; Microsoft Docs dives into far greater detail: https://docs.microsoft.com/dotnet/api/system.object.gethashcode#remarks.

This example extends the existing class that supports `IComparable` to support `IEquatable`. Note that `IEquatable` uses `[object]` instead of `[MyClass]`:

```
class MyClass : IComparable, IEquatable[object] {
    [int] $Number
    [int] CompareTo([object] $object) {
        if ($this.Number -gt $object.Number) {
            return 1
        } elseif ($this.Number -lt $object.Number) {
            return -1
        } else {
            return 0
        }
    }
    [int] GetHashCode() {
        return $this.Number
    }
    [bool] Equals(
        [object] $equalTo
    ) {
        return $this.Number -eq $equalTo.Number
    }
}
```

Note that the argument type for the `Equals` method must be `[object]` in the preceding example; it must match the type used in `IEquatable[object]`.

Once the interface has been implemented, two instances of the class may be compared using the `-eq` operator.

```
PS> $first = [MyClass]@{ Number = 1 }
PS> $second = [MyClass]@{ Number = 1 }
PS> $first -eq $second
True
```

PowerShell lacks language support to implement every available interface; in some cases it is possible to work around the limitations, as is the case with `IEquatable`. In others, such as `IFormattable`, PowerShell is simply not capable enough yet.

Supporting casting between value types requires another technique.

Supporting casting

The simplest way to support casting between PowerShell classes is to create a constructor that will accept the incoming type.

For example, it is possible to cast from `Int` to `MyClass` by implementing a constructor that accepts `Int` as an argument.

```
class MyClass {
    [int] $Number
    MyClass() { }
    MyClass([int] $value) {
        $this.Number = $value
    }
}
```

With the constructor in place, PowerShell will allow `Int32` values to be cast to the class:

```
PS> [MyClass]1
Number
------
     1
```

If PowerShell can convert a value into `Int32`, then the assignment will succeed. Therefore `short` (`Int16`), `byte`, and other values smaller than `Int32` will succeed.

Attempting to cast larger types, `long` (`Int64`), will fail:

```
PS> [MyClass][long]1
InvalidArgument: Cannot convert the "1" value of type "System.Int64" to type "MyClass".
```

PowerShell is also capable of coercing a value to a type using a `Parse` static method. In the following example, the method accepts any object, and if it can coerce that to `Int32`, it will return an instance of `MyClass`:

```
class MyClass {
    [int] $Number

    static [MyClass] Parse(
        [object] $value
    ) {
        if ($value -as [int]) {
            return [MyClass]@{ Number = $value }
        } else {
            throw 'Unsupported value'
        }
    }
}
```

As this version parses values of type `[object]` (or anything) it will accept larger value types, if the actual value can be reduced into an `Int32`:

```
PS> [MyClass][long]2312
Number
------
  2312
```

The preceding methods support casting from a specific value to an instance of `MyClass`. It is possible for the class to support casting to another type by implementing an implicit conversion operator. The operator is implemented as a static method named `op_Implicit`:

```
class NewClass {
    [string] $DayOfWeek
}
class MyClass {
    [int] $Number
    hidden static [NewClass] op_Implicit(
        [MyClass] $instance
    ) {
        return [NewClass]@{
            DayOfWeek = [DayOfWeek]$instance.Number
        }
    }
}
```

The advantage of this approach is that the conversion logic is part of `MyClass`. `NewClass` does not have to know anything about `MyClass` or how to perform the conversion:

```
PS> [NewClass][MyClass]@{ Number = 1 }
DayOfWeek
---------
Monday
```

The disadvantage of this approach is that a maximum of two `op_Implicit` methods can be added. Each method requires a unique signature (name and arguments), and the return type is not part of that signature.

You can add a second signature by changing the argument type to `[object]`; in this case it supports casting to the `[DayOfWeek]` enumeration:

```
class NewClass {
    [string] $DayOfWeek
}
class MyClass {
    [int] $Number
    hidden static [NewClass] op_Implicit(
        [MyClass] $instance
    ) {
        return [NewClass]@{
            DayOfWeek = [DayOfWeek]$instance.Number
        }
    }
    hidden static [DayOfWeek] op_Implicit(
        [object] $instance
    ) {
        return [DayOfWeek]$instance.Number
    }
}
```

Classes can be used for a variety of existing purposes in PowerShell. Classes can be used to implement parameter transformation and validation.

Classes for parameters

PowerShell includes several attributes that you can use on parameters. Each of these classes inherits from another .NET type. The classes include:

- Argument transformation attributes
- Validation attributes
- Classes for use with `ValidateSet`

You can implement each of these using classes in PowerShell.

Argument transformation attributes are used to test and potentially change the value being assigned to a variable before the assignment completes.

Argument-transformation attribute classes

An argument-transformation attribute is used to convert the value of an argument for a variable. The transformation operation is carried out before PowerShell completes the assignment to the variable, giving the opportunity to avoid type mismatch errors.

You can add argument-transformation attributes to any variable, including parameters in functions and scripts and properties in classes.

Classes for argument transformation must implement a `Transform` method. The `Transform` method must accept two arguments, a `System.Object` and a `System.Management.Automation.EngineIntrinsics`. The argument names are arbitrary; the names used follow the example on Microsoft Docs: https://docs.microsoft.com/dotnet/api/system.management.automation.argumenttransformationattribute.transform.

The class must therefore inherit from `System.Management.Automation.ArgumentTransformationAttribute`.

Abstract classes like the `ArgumentTransformationAttribute` type are somewhat like interfaces. However, where a class may implement one or more interfaces, a class can only inherit from one abstract type. In essence this ensures that the class implements the required functionality.

The following example is an argument-transformation attribute that converts a date string in the format `yyyyMMddHHmmss` to `DateTime` before the assignment to the parameter is completed:

```
using namespace System.Management.Automation
class DateTimeStringTransformationAttribute :
    ArgumentTransformationAttribute {
    [object] Transform(
        [EngineIntrinsics] $engineIntrinsics,
        [object]           $inputData
    ) {
        if ($inputData -is [DateTime]) {
            return $inputData
        }
        if ($inputData -is [string]) {
            $date = Get-Date
            $isValidDate = [DateTime]::TryParseExact(
                $inputData,
                'yyyyMMddHHmmss',
                $null,
                'None',
                [ref]$date
            )
            if ($isValidDate) {
                return $date
            }
        }
        throw 'Unexpected date format'
    }
}
```

The class does not need to contain anything other than the `Transform` method implementation. If the transformation is more complex, it may implement other helper methods. The following example moves `[DateTime]::TryParseExact` into a new method:

```
using namespace System.Management.Automation
class DateTimeStringTransformationAttribute :
    ArgumentTransformationAttribute {
    hidden [DateTime] $date
    hidden [bool] tryParseExact(
        [string] $value
    ) {
        $parsedDate = Get-Date
        $parseResult = [DateTime]::TryParseExact(
            $value,
            'yyyyMMddHHmmss',
            $null,
            'None',
            [Ref]$parsedDate
        )
        $this.date = $parsedDate
        return $parseResult
    }
    [object] Transform(
        [EngineIntrinsics] $engineIntrinsics,
        [object]           $inputData
    ) {
        if ($inputData -is [DateTime]) {
            return $inputData
        }
        if ($inputData -is [string]) {
            if ($this.tryParseExact($inputData)) {
                return $this.date
            }
        }
        throw 'Unexpected date format'
    }
}
```

The new class may be used with a parameter, as shown here. Note that the `Attribute` string at the end of the class name may be omitted when it is used:

```
function Test-Transform {
    param (
        [DateTimeStringTransformation()]
        [DateTime] $Date
    )

    Write-Host $Date
}
```

With this attribute in place, the function can be passed a date and time in a format that would not normally convert:

```
PS> Test-Transform -Date '20210102090000'
02/01/2021 09:00:00
```

Classes can also be used to implement customized validation.

Validation attribute classes

You can use PowerShell classes to build custom validation attributes. This offers an alternative to `ValidateScript`.

Validation attributes must inherit from either `ValidateArgumentsAttribute` or `ValidateEnumeratedArgumentsAttribute`.

Validators are most often used with parameters in scripts and functions, but they may be used with any variable.

ValidateArgumentsAttribute

Validators that inherit from `ValidateArgumentsAttribute` are somewhat difficult to define. The existing validators, such as `ValidateNotNullOrEmpty` and `ValidateCount`, catch most of the possible uses. Validation is more often interested in testing whether the value of a parameter is an array.

Classes that inherit from `ValidateArgumentsAttribute` act on an argument as a single entity. If an argument is an array, the validation step applies to the array object rather than the individual elements of the array.

Classes that implement `ValidateArgumentsAttribute` must inherit from `System.Management.Automation.ValidateArgumentsAttribute`. The class must implement a `Validate` method that is marked as abstract in the `ValidateArgumentsAttribute` class.

The `Validate` method accepts two arguments with the `System.Object` and `System.Management.Automation.EngineIntrinsics` types. This is shown on Microsoft Docs: https://docs.microsoft.com/dotnet/api/system.management.automation.validateargumentsattribute.validate.

The following example tests that the argument is not null or whitespace:

```
using namespace System.Management.Automation
class ValidateNotNullOrWhitespaceAttribute :
    ValidateArgumentsAttribute {
    [void] Validate(
        [object]           $arguments,
        [EngineIntrinsics] $engineIntrinsics
    ) {
        if ([String]::IsNullOrWhitespace($arguments)) {
            throw 'The value cannot be null or white space'
        }
    }
}
```

The attribute is placed on top of a parameter and will run for every assignment statement:

```
function Test-Validate {
    [CmdletBinding()]
    param (
        [ValidateNotNullOrWhitespace()]
        [String]$Value
    )

    Write-Host $Value
}
```

If an empty string is passed as an argument for the `Value` parameter, an error will be displayed:

```
PS> Test-Validate -Value '   '
Test-Validate: Cannot validate argument on parameter 'Value'. The value cannot be null or white space
```

This validator example will be more effective when defined as a validator that inherits from `ValidateEnumeratedArguments` and is implemented in the `ValidateElement` method.

ValidateEnumeratedArgumentsAttribute

Classes that inherit from `ValidateEnumeratedArgumentsAttribute` may be used to test each of the elements in an array (when associated with an array-based parameter), or a single item (when associated with a scalar parameter).

Classes that implement `ValidateEnumeratedArgumentsAttribute` must inherit from `System.Management.Automation.ValidateEnumeratedArgumentsAttribute`. The class must implement a `Validate` method that is marked as abstract in the `ValidateEnumeratedArgumentsAttribute` class.

The `ValidateElement` method accepts one argument with the `System.Object` type. This is shown on Microsoft Docs: https://docs.microsoft.com/dotnet/api/system.management.automation.validateenumeratedargumentsattribute.validateelement.

The `ValidateElement` method does not return any output; it either runs successfully or throws an error. The error will be displayed to the end user.

The next code block validates that an IP address used as an argument falls in a private address range.

There are many different methods for determining if an address range is private. The following class tests the first 3 octets of the IP address. For instance, if the first octet is 10, the address is private. If the first octet is 172, and the second is between 16 and 31, the address is private, and so on.

If the address is not part of a private range, or not a valid IP address, the `ValidateElement` method will throw an error:

```
using namespace System.Management.Automation
class ValidatePrivateIPAddressAttribute :
        ValidateEnumeratedArgumentsAttribute {
    hidden [bool] IsPrivateIPAddress(
        [IPAddress] $address
    ) {
        $bytes = $address.GetAddressBytes()
        $isPrivateIPAddress = switch ($null) {
            { $bytes[0] -eq 10 } { $true; break }
            { $bytes[0] -eq 172 -and
              $bytes[1] -ge 16 -and
              $bytes[1] -le 31 } { $true; break }
            { $bytes[0] -eq 192 -and
              $bytes[1] -eq 168 } { $true; break }
            default { $false }
        }
        return $isPrivateIPAddress
    }
    [void] ValidateElement(
        [object] $element
    ) {
        $ipAddress = $element -as [IPAddress]
        if (-not $ipAddress) {
             throw '{0} is an invalid IP address format' -f @(
                $element
            )
        }
        if (-not $this.IsPrivateIPAddress($element)) {
            throw '{0} is not a private IP address' -f @(
                $element
            )
        }
    }
}
```

You can use the preceding attribute with any parameter to validate IP addressing, as shown in the following short function:

```
function Test-Validate {
    [CmdletBinding()]
    param (
        [ValidatePrivateIPAddress()]
        [IPAddress]$IPAddress
    )

    Write-Host $IPAddress
}
```

When used, the function will only allow the address as an argument for the parameter if it is in one of the reserved private

ranges:

```
PS> Test-Validate -IPAddress 10.0.0.11
10.0.0.11
```

When a public IP address is specified, binding will fail:

```
PS> Test-Validate -IPAddress 50.0.0.11
Test-Validate: Cannot validate argument on parameter 'IPAddress'. 50.0.0.11 is not a private IP
address
```

If something other than an IP address is supplied, the error will state the IP address is invalid:

```
PS> Test-Validate -IPAddress someString
Test-Validate: Cannot process argument transformation on parameter 'IPAddress'. Cannot convert value
"someString" to type "System.Net.IPAddress". Error: "An invalid IP address was specified."
```

Validation like this can be implemented with `ValidateScript`, which also inherits from `ValidateEnumeratedArgumentsAttribute`. `ValidateScript` can call functions, centralizing the validation code where necessary.

Classes for ValidateSet

`ValidateSet` in Windows PowerShell will test the value assigned to a variable (often a parameter) against a hard-coded list of values.

In PowerShell 7, you can use `ValidateSet` with a class to dynamically determine the values to validate against. This is achieved by writing a class that implements the `IValidateSetValuesGenerator` interface: https://docs.microsoft.com/dotnet/api/system.management.automation.ivalidatesetvaluesgenerator.

The interface requires the class to implement a `GetValidValues` method, which should return an array of strings. The class can use any method it needs to generate the list of values. In the following example, the list is derived from the list of environment variable names in the current process.

```
using namespace System.Management.Automation
class EnvironmentVariable : IValidateSetValuesGenerator {
    [string[]] GetValidValues() {
        return Get-ChildItem env: -Name
    }
}
```

You can use the class for tab completion and to validate the values for a parameter of a function:

```
function Get-EnvironmentVariable {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory)]
        [ValidateSet([EnvironmentVariable])]
        [string]$Name
    )
    Get-Item env:$Name
}
```

The preceding classes are mostly used with parameters, but you can use them with any variable, including properties in classes.

Microsoft Desired State Configuration is perhaps the most important use for classes in PowerShell.

Classes and DSC

Microsoft **DSC**, or **Desired State Configuration**, is one of several different configuration management systems available. Individual items are configured idempotently, that is, they only change when change is required.

Classes in PowerShell exist because of DSC. DSC resources written as PowerShell classes are very succinct; they avoid the repetition inherent in a script-based resource. Script-based resources must at least duplicate a `param` block.

Class-based DSC resources in a module must be explicitly exported using the `DscResourcesToExport` key in a module manifest document.

The class must include a `DscResource` attribute. Each property a user is expected to set must have a `DscProperty` attribute. At least one property must be the `Key` property of the `DscProperty` attribute set. The class must implement the `Get`, `Set`, and `Test` methods.

Class-based resources may use inheritance to simplify an implementation as required; this is especially useful if a group of resources uses the same code to act out changes.

A basic DSC resource is defined as follows:

```
enum Ensure {
    Absent
    Present
}
[DscResource()]
class MyResource {
    [DscProperty(Key)]
    [Ensure] $Ensure
    [MyResource] Get() { return $this }
    [void] Set() { }
    [bool] Test() { return $true }
}
```

This resource implements all the required methods, but it performs no actions.

Like a good function, a good DSC resource should strive to be good at one thing and one thing only. If a particular item has a variety of configuration options, it is often better to have a set of similar resources than a single resource that attempts to do it all.

The sections that follow will focus on the creation of a short resource that sets the computer description.

This resource will need to make a change to a single registry value. The computer description is set under the `HKLM:\SYSTEM\CurrentControlSet\Services\LanmanServer\Parameters` key using the `svrcomment` string value.

The starting point for the resource is shown here:

```
enum Ensure {
    Absent
    Present
}
[DscResource()]
class ComputerDescription {
    [DscProperty(Key)]
    [Ensure]$Ensure
    [DscProperty()]
    [string]$Description
    $path = 'HKLM:\SYSTEM\CurrentControlSet\Services\LanmanServer\Parameters'
    $valueName = 'svrcomment'
    [ComputerDescription] Get() { return $this }
    [void] Set() { }
    [bool] Test() { return $true }
}
```

Each of the methods in the class must be implemented for the resource to function.

Implementing Get

The `Get` method should evaluate the current state of the resource. The registry key will exist, but the registry value may be incorrect, or may not exist.

The `Get` method will act as follows:

- If the value is present, it will set the `Ensure` property to `Present` and update the value of the `Description` property
- If the value is not present, it will set the `Ensure` property to `Absent` only

The following snippet implements these actions:

```
class ComputerDescription {
    [ComputerDescription] Get() {
        $key = Get-Item $this.Path
        if ($key.GetValueNames() -contains $this.valueName) {
            $this.Ensure = 'Present'
            $this.Description = $key.GetValue($this.valueName)
        } else {
            $this.Ensure = 'Absent'
        }
        return $this
    }
}
```

The `Get` method must return an instance of the class. It can either return the existing instance, return `$this`, or generate a new instance, for instance by returning a `hashtable`:

```
class ComputerDescription {
    [ComputerDescription] Get() {
        $properties = @{}
        $key = Get-Item $this.Path
        if ($key.GetValueNames() -contains $this.valueName) {
            $properties.Ensure = 'Present'
            $properties.Description = $key.GetValue($this.valueName)
        } else {
            $properties.Ensure = 'Absent'
        }
        return $properties
    }
}
```

The `hashtable` returned by the preceding function is automatically cast to the class, creating a new instance.

The `Get` method is only used when explicitly invoked. It is not used by either `Set` or `Test`.

Implementing Set

The `Set` method deals with making a change if a change is required. `Set` can ordinarily assume that `Test` has been run, and therefore that a change is required.

As the resource allows a user to ensure a value is either present or absent, it must handle the creation and deletion of the value:

```
class ComputerDescription {
    [Void] Set() {
        $commonParams = @{
            Path = $this.path
            Name = $this.valueName
        }
        if ($this.Ensure -eq 'Present') {
            $newParams = @{
                Value = $this.Description
                Type  = 'String'
                Force = $true
            }
            New-ItemProperty @newParams @commonParams
        } else {
            $key = Get-Item $this.Path
            if ($key.GetValueNames() -contains $this.valueName) {
                Remove-ItemProperty @commonParams
            }
        }
    }
}
```

This version of `Set` uses the `Force` parameter of `New-ItemProperty` to overwrite any existing values of the same name. Using `Force` also handles cases where the value exists, but the value type is incorrect.

Implementing Test

You can use the `Test` method to determine whether `Set` should be run. DSC invokes `Test` before `Set`. The `Test` method returns a Boolean value.

The `Test` method must perform the following tests to ascertain the state of this configuration item:

- When `Ensure` is present, fail if the value does not exist
- When `Ensure` is present, fail if the value exists, but the description does not match the requested value
- When `Ensure` is absent, fail if the value name exists
- Otherwise, pass

The following snippet implements these tests:

```
class ComputerDescription {
    [bool] Test() {
        $key = Get-Item $this.Path
        if ($this.Ensure -eq 'Present') {
            if ($key.GetValueNames() -notcontains $this.valueName) {
                return $false
            }
            return $key.GetValue($this.valueName) -eq
                $this.Description
        } else {
            return $key.GetValueNames() -notcontains $this.valueName
        }
        return $true
    }
}
```

Each of these methods must be copied back into the resource class.

Using the resource

The complete class, `ComputerDescription`, incorporating each of the preceding methods, is shown here:

```
enum Ensure {
    Absent
    Present
}
[DscResource()]
class ComputerDescription {
    [DscProperty(Key)]
    [Ensure] $Ensure
    [DscProperty()]
    [string] $Description
    $path = 'HKLM:\SYSTEM\CurrentControlSet\Services\LanmanServer\Parameters'
    $valueName = 'svrcomment'
    [ComputerDescription] Get() {
        $key = Get-Item $this.Path
        if ($key.GetValueNames() -contains $this.valueName) {
            $this.Ensure = 'Present'
            $this.Description = $key.GetValue($this.valueName)
        } else {
            $this.Ensure = 'Absent'
        }
        return $this
    }
    [void] Set() {
        $commonParams = @{
            Path = $this.path
            Name = $this.valueName
        }
        if ($this.Ensure -eq 'Present') {
            $newParams = @{
                Value = $this.Description
                Type  = 'String'
                Force = $true
            }
            New-ItemProperty @newParams @commonParams
        } else {
            $key = Get-Item $this.Path
            if ($key.GetValueNames() -contains $this.valueName) {
                Remove-ItemProperty @commonParams
            }
        }
    }
    [bool] Test() {
        $key = Get-Item $this.Path
        if ($this.Ensure -eq 'Present') {
            if ($key.GetValueNames() -notcontains $this.valueName) {
                return $false
            }
            return $key.GetValue($this.valueName) -eq
                $this.Description
        } else {
            return $key.GetValueNames() -notcontains
            $this.valueName
        }
        return $true
    }
}
```

DSC will only find the class using `Get-DscResource` if the following are true:

- The class is saved in a module
- The module exports the DSC resource
- The module is in one of the paths in `$env:PSMODULEPATH`
- The module path is system-wide, accessible by the **Local Configuration Manager** (**LCM**)

The following script creates the files and folders required to achieve under `Program Files`. The script will require administrative rights:

```
$modulePath = 'C:\Program Files\WindowsPowerShell\Modules'
$newItemParams = @{
    Path     = Join-Path $modulePath 'LocalMachine\1.0.0\LocalMachine.psm1'
    ItemType = 'File'
    Force    = $true
}
New-Item @newItemParams
$joinPathParams = @{
    Path      = $modulePath
    ChildPath = 'LocalMachine\1.0.0\LocalMachine.psd1'
}
$newModuleManifestParams = @{
    Path                 = Join-Path @joinPathParams
    RootModule           = 'LocalMachine.psm1'
    DscResourcesToExport = 'ComputerDescription'
}
New-ModuleManifest @newModuleManifestParams
```

The `LocalMachine.psm1` file should be edited, adding the `Ensure` enumeration and the `ComputerDescription` class.

Once the class is in a module, it can be used with the `using module` command:

```
using module LocalMachine
$class = [ComputerDescription]@{
    Ensure      = 'Present'
    Description = 'Computer description'
}
```

Individual methods may be invoked; for example, you can run `Get`:

```
PS> $class.Get()
Ensure    Description
------    -----------
Absent    Computer description
```

As the module is under a known module path, `Get-DscResource` should be able to find it immediately:

```
PS> Get-DscResource ComputerDescription
ImplementedAs    Name                   ModuleName      Version
-------------    ----                   ----------      -------
PowerShell       ComputerDescription    LocalMachine    1.0
```

You can use the `Invoke-DscResource` command to run individual methods without creating a DSC configuration document:

```
$params = @{
    Name       = 'ComputerDescription'
    ModuleName = 'LocalMachine'
    Method     = 'Test'
    Property   = @{
        Ensure      = 'Present'
        Description = 'Some description'
    }
    Verbose    = $true
}
Invoke-DscResource @params
```

Running `Invoke-DscResource` will require administrative rights. `Invoke-DscResource` interacts with the LCM to execute the

resource and will report back whether the configuration item is in the desired state.

Summary

Enumerations in PowerShell are useful for defining customized lists of constant values for use within a script, function, or module. The values in the enumeration can be used if required, or the enumeration can be used as little more than a list of names.

Flag-based enumerations are used when managing flag-based fields. The `FileSystemRights` enumeration was used as a basis for demonstrating capabilities.

You can use classes in PowerShell to describe any object. Classes include members such as properties and methods. The `Hidden` keyword can be used to hide either properties or methods from view.

Class inheritance is a vital part of working with classes. One class can inherit properties and methods from another, allowing a layered approach to class implementation with shared code in underlying classes.

Inheritance-style notation is the basis for implementing interfaces from .NET in a class in PowerShell. The `IComparable` and `IEquatable` interfaces and the operator support they add were demonstrated.

Casting and coercing types in PowerShell is a complex process. Constructors and the `Parse` static method allow PowerShell to cast from a fixed type to a class defined in PowerShell. The implicit conversion operator can be implemented as a static method named `op_Implicit` in a PowerShell class to allow casting to one or two other types.

You can use classes in PowerShell to implement custom argument transformation and validation attributes. The validation and transformation classes are derived from abstract classes in the `System.Management.Automation` namespace.

In PowerShell 6, `ValidateSet` gained the ability to use a dynamic set provided by a class implementing the `IValidateSetValuesGenerator` interface. The creation and use of the validator were demonstrated.

Microsoft Desired State Configuration is one of the driving forces behind the existence of classes in PowerShell. A simple DSC resource that allows a computer description to be set via the registry was created and demonstrated.

In the next chapter, you will build modules in PowerShell.

20

Building Modules

A module groups a set of commands together, most often around a common system, service, or purpose. Modules were introduced with PowerShell 2.

A module written in PowerShell can contain functions, classes, enumerations, and DSC resources, and it may define aliases and include variables to include in the global scope.

There are several different common styles or layouts used when authoring modules. This chapter explores these different approaches, starting with the simplest.

This chapter explores the following topics:

- Creating a module
- Publishing a module
- Multi-file module layout
- Module scope
- Initializing and removing modules

Technical requirements

This chapter uses the following modules, which can be installed from the PowerShell Gallery:

- PowerShellGet version 2.2.5
- ModuleBuilder version 2.0.0

The module created in this chapter is available on GitHub: https://github.com/PacktPublishing/Mastering-Windows-PowerShell-Scripting-Fourth-Edition/tree/master/Chapter20/LocalMachine.

The repository includes the different layouts discussed in this chapter.

Creating a module

A module most often consists of one or more functions. Typically, you create modules in one or more files starting with a `psm1` file. The `psm1` file is known as the **root module**.

The root module

The root module has a `psm1` extension and is otherwise like any other script file in PowerShell. The root module file is named after the module.

The `psm1` file can contain all the module content directly; nothing else is required to create a module.

A module requires content. *Chapter 19, Classes and Enumerations*, ended by creating a class-based DSC resource to manage the computer description property. You'll rebuild this content of the resource and use it as the basis for creating a module during this chapter.

Here is the first command to add to a file named `LocalMachine.psm1`:

```
function Get-ComputerDescription {
    [CmdletBinding()]
    [OutputType([string])]
    param ( )
    $getParams = @{
        Path = 'HKLM:\SYSTEM\CurrentControlSet\Services\LanmanServer\Parameters'
        Name = 'srvcomment'
    }
    Get-ItemPropertyValue @getParams
}
```

Once saved, you can import the module file. The following command assumes the file is in the current directory in PowerShell:

```
Import-Module .\LocalMachine.psm1
```

The command will be displayed in the `ExportedCommands` property when running `Get-Module`:

```
PS> Get-Module LocalMachine
ModuleType Version PreRelease Name         ExportedCommands
---------- ------- ---------- ----         ----------------
Script     0.0                LocalMachine Get-ComputerDescription
```

The `ModuleType` shows as `Script` because the module is defined by a `psm1` file. If `ExportedCommands` is empty, there may be an error in the module file. Attempting to import the module should show any errors.

You can run the command if the module is imported. If a computer description is set, it will be displayed, otherwise an error will be displayed. You can optionally set a description by running the following command as administrator:

```
$params = @{
    Path  = 'HKLM:\SYSTEM\CurrentControlSet\Services\LanmanServer\Parameters'
    Name  = 'srvcomment'
    Value = 'Computer description'
}
New-ItemProperty @params
```

As you add more functions to the `psm1` file, additional commands will be displayed in `ExportedCommands`. By default, all functions are exported from the module, making them available to anyone using the module.

Export-ModuleMember

You can use the `Export-ModuleMember` command inside a `psm1` file to explicitly define what is exported from a module.

At this point, the `LocalMachine` module only contains one command, and that command is expected to be visible to the end user.

As the module grows, some of the code will start to repeat, and functions might be added to the module to reduce repetition. Such functions might be expected to be internal to the module, not visible to a user of the module.

Functions that are exported are often referred to as public functions; functions that are not exported are known as private functions.

The following change to `LocalMachine.psm1` introduces a function that will be shared inside the module and is not intended to be seen by users of the module.

The private function anticipates that the registry path and value name parameters will be used by other functions in the module later:

```
function GetRegistryParameter {
    [CmdletBinding()]
    [OutputType([Hashtable])]
    param ( )
    @{
        Path = 'HKLM:\SYSTEM\CurrentControlSet\Services\LanmanServer\Parameters'
        Name = 'srvcomment'
    }
}
```

The private function `GetRegistryParameter` is used by the `Get-ComputerDescription` command, as the following shows:

```
function Get-ComputerDescription {
    [CmdletBinding()]
    [OutputType([string])]
    param ( )
    $getParams = GetRegistryParameter
    Get-ItemPropertyValue @getParams
}
```

`GetRegistryParameter` is also used by both `Set-ComputerDescription` and `Remove-ComputerDescription`. The short `Remove-ComputerDescription` function is:

```
function Remove-ComputerDescription {
    [CmdletBinding(SupportsShouldProcess)]
    param ( )
    $removeParams = GetRegistryParameter
    if ($PSCmdlet.ShouldProcess(
        'Removing computer description')) {
        Remove-ItemProperty @removeParams
    }
}
```

And finally, you can add the `Set-ComputerDescription` command to complete the functions for this version of the module. Note that `Set-ComputerDescription` requires administrator rights to execute:

```
function Set-ComputerDescription {
    [CmdletBinding(SupportsShouldProcess)]
    [OutputType([string])]
    param (
        [Parameter(Mandatory, Position = 1, ValueFromPipeline)]
        [string]$Description
    )
    process {
        if ($pscmdlet.ShouldProcess(
            'Removing computer description')) {
            $setParams = GetRegistryParameter
            $params = @{
                Type  = 'String'
                Value = $Description
                Force = $true
            }
            try {
                New-ItemProperty @setParams @params
            } catch {
                $PSCmdlet.ThrowTerminatingError($_)
            }
        }
    }
}
```

The `Set-ComputerDescription` command adds a few features first explored in *Chapter 17, Scripts, Functions, and Script Blocks*. First it has a mandatory parameter for the new description. That `Description` parameter accepts values from the pipeline.

Both the `Remove-ComputerDescription` and `Set-ComputerDescription` commands implement `SupportsShouldProcess`, enabling the use of the `-WhatIf` switch parameter. Implementing `SupportsShouldProcess` is an encouraged practice for commands that change state.

If the functions are saved to `LocalMachine.psm1`, and the module is imported, then the new command will show in `ExportedCommands`:

```
PS> Import-Module .\LocalMachine.psm1 -Force -Verbose
VERBOSE: Loading module from path 'C:\workspace\LocalMachine.psm1'.
VERBOSE: Importing function 'Get-ComputerDescription'.
VERBOSE: Importing function 'Remove-ComputerDescription'.
VERBOSE: Importing function 'Set-ComputerDescription'.
PS> Get-Module -Name LocalMachine |
>>     ForEach-Object ExportedCommands
Key                        Value
---                        -----
Get-ComputerDescription    Get-ComputerDescription
GetRegistryParameters      GetRegistryParameters
Remove-ComputerDescription Remove-ComputerDescription
Set-ComputerDescription    Set-ComputerDesciption
```

You used the `Force` parameter with `Import-Module` in the previous example to overwrite the previously imported version of the module. If a module is already imported, `Import-Module` will not, by default, reimport that module.

Naming private commands

`GetRegistryParameters` follows my own convention for naming private commands. Private commands always use an approved verb (see `Get-Verb`) but omit the hyphen.

This convention is not mandatory; the naming of private commands is a personal preference.

Adding the `Export-ModuleMember` command to the end of the `LocalMachine.psm1` file will hide the `GetRegistryParameters` function from view:

```
Export-ModuleMember -Function @(
    'Get-ComputerDescription'
    'Remove-ComputerDescription'
    'Set-ComputerDescription'
)
```

Importing the module again shows the change to `ExportedCommands`:

```
PS> Import-Module .\LocalMachine.psm1 -Force
PS> Get-Module -Name LocalMachine |
>>     ForEach-Object ExportedCommands
Key                        Value
---                        -----
Get-ComputerDescription    Get-ComputerDescription
Remove-ComputerDescription Remove-ComputerDescription
Set-ComputerDescription    Set-ComputerDesciption
```

`Export-ModuleMember` can be used to define which functions, cmdlets, variables, and aliases are exported from a module. You can also define these options by using a module manifest.

Module manifests

The module manifest is a PowerShell data file (`psd1` file, a Hashtable stored in a file) that contains metadata for a module.

The manifest includes information such as the module description, PowerShell version and edition compatibility, version number, commands, aliases, variables that should be exported, and so on.

Help for either the `New-ModuleManifest` or `Update-ModuleManifest` commands will show the possible keys. A manifest created using `New-ModuleManifest` includes comments describing the purpose of each field.

The use of a manifest is mandatory if a module is published on the PowerShell Gallery or any other repository when using the `Publish-Module` command.

The following code creates a module manifest for the `LocalMachine` module:

```
$params = @{
    Path                 = '.\LocalMachine.psd1'
    Description          = 'Local machine configuration'
    RootModule           = 'LocalMachine.psm1'
    ModuleVersion        = '1.0.0'
    FunctionsToExport    = @(
        'Get-ComputerDescription'
        'Remove-ComputerDescription'
        'Set-ComputerDescription'
    )
    PowerShellVersion    = '5.1'
    CompatiblePSEditions = @(
        'Core'
        'Desktop'
    )
}
New-ModuleManifest @params
```

In the preceding example, `RootModule` is the full name of the `psm1` file. This path, and all other paths used in the manifest, are relative to the module manifest file.

`RootModule` is used to determine the module type displayed when running the `Get-Module` command:

- If `RootModule` is set to a `psm1` file (as in our example), the module type will be `script`
- If `RootModule` is set to a `dll` file, the module type will be `binary`
- If `RootModule` is not set or is set to a value with any other file extension, the module type will be `manifest`

The `RootModule` property is not the only value that affects the module type. A module in PowerShell can have other modules nested inside. Nested modules are loaded via a `NestedModules` property in the manifest:

- If `NestedModules` is set, the module type will be `manifest` (regardless of the `RootModule` value)

Other than affecting the module type value, `NestedModules` is beyond the scope of this chapter.

The newly created module manifest will, by default, set `CmdletsToExport`, `VariablesToExport`, and `AliasesToExport` to `*`. You can set these values to `@()` if automatically exporting everything is not desirable.

You can use wildcards for any of the `ToExport` properties, including `FunctionsToExport`. For example, you could use the value `*-*` to export public functions in the `LocalMachine` module. Using a specific list helps the module autoloader in PowerShell import the module if one of the commands is used and the module has never been imported before.

This manifest replaces the functionality of the `Export-ModuleMember` command.

PowerShell and PowerShellGet include several commands for working with the content of the module manifest.

Test-ModuleManifest

The `Test-ModuleManifest` command attempts to read and perform basic checks of the values used in the module manifest.

The command returns an object that represents data in the manifest. `Import-PowerShellDataFile` can also be used to read the manifest if required.

If the `PowerShellVersion` property in the manifest is reduced to `5.0` in `LocalMachine.psd1`, `Test-ModuleManifest` will raise an error pointing out that the `CompatiblePSEditions` key is not available before PowerShell 5.1:

```
PS> Test-ModuleManifest .\LocalMachine.psd1
Test-ModuleManifest: The module manifest 'C:\Workspace\LocalMachine\LocalMachine\LocalMachine.psd1' is
specified with the CompatiblePSEditions key which is supported only on PowerShell version '5.1' or
higher. Update the value of the PowerShellVersion key to '5.1' or higher, and try again.
ModuleType Version PreRelease Name         ExportedCommands
---------- ------- ---------- ----         ----------------
Script     1.0.0              LocalMachine Get-ComputerDescription
```

`Test-ModuleManifest` will also test paths used in the manifest. For example, if an absolute path is used for `RootModule`, such as `C:\LocalMachine.psm1`, an error will be displayed. The `RootModule` path must be a relative path.

The `New-ModuleManifest` command can, but should not, be used to update the content of an existing manifest. Instead, you should use the `Update-ModuleManifest` command.

Update-ModuleManifest

You can use the `Update-ModuleManifest` command from the PowerShellGet module to update the content of an existing manifest. However, you cannot use `Update-ModuleManifest` to create a new manifest.

The advantage of using `Update-ModuleManifest` is that it correctly handles writing to the `PrivateData` section of the manifest. `New-ModuleManifest` is unable to set values in this section.

You can use `Update-ModuleManifest` to add a project URL and license information to the manifest, as shown here:

```
Update-ModuleManifest -Path .\LocalMachine.psd1 -PrivateData @{
    ProjectUri = 'https://github.com/indented-automation/LocalMachine'
    LicenseUri = 'https://opensource.org/licenses/MIT'
}
```

Similarly, you can update the version number in the manifest; in this case, the major version is updated:

```
$path = '.\LocalMachine.psd1'
$manifest = Test-ModuleManifest -Path $path
$newVersion = [Version]::new(
     $manifest.Version.Major + 1,
     $manifest.Version.Minor,
     $manifest.Version.Build
)
Update-ModuleManifest -Path $path -ModuleVersion $newVersion
```

You may want to publish the module to a PowerShell repository at this point.

Publishing a module

Use the `Publish-Module` command to publish a module to a PowerShell repository.

The most common repository types are a NuGet feed (like the PowerShell Gallery), or a directory (often a file share). There are several free options available for creating a dedicated private PowerShell repository, including:

- Chocolatey Server: https://community.chocolatey.org/packages/chocolatey.server
- Nexus OSS: https://www.sonatype.com/products/repository-oss-download
- ProGet: https://inedo.com/proget/pricing

Each of these allows you to create a private repository for use within an organization. The repository would typically contain internally developed content or curated content that has been approved for internal use copied from public repositories.

To test publishing the `LocalMachine` module, create a repository in a local directory:

```
New-Item ~\PSLocal -ItemType Directory
Register-PSRepository -Name PSLocal -Source ~\PSLocal
```

Repositories such as the PowerShell Gallery require an API key to authenticate the request.

Removing the PSLocal repository

When it is no longer required, the repository can be removed:

```
Unregister-PSRepository -Name PSLocal
```

You can now publish the module with the `Publish-Module` command:

```
Publish-Module -Path . -Repository PSLocal
```

The `Path` used with `Publish-Module`, the current working directory in the preceding example, has a constraint: the path must be a directory and it must be named after the module.

That is, if the module is `LocalMachine.psd1`, and the manifest's parent directory is named `LocalMachine`, then publishing will likely succeed, showing a directory structure like the following:

```
ProjectRoot
| -- LocalMachine
     | -- LocalMachine.psd1
     | -- LocalMachine.psm1
```

Once published, a `nupkg` file will be created in the `PSLocal` directory, like in this example:

```
PS> Get-ChildItem ~\PSLocal
        Directory: C:\Users\Chris\PSLocal
Mode                LastWriteTime  Length Name
----                -------------  ------ ----
-a---        11/04/2021     12:00    3856 LocalMachine.2.0.0.nupkg
```

A versioned directory is also permitted, provided the parent of that is named after the module. This is part of the side-by-side versioning support introduced with PowerShell 5.0.

Publishing and side-by-side versioning

Before PowerShell 5, only one version of a module could be installed under each module path. The content of a `Modules` directory was as shown below, which is like the structure used in the project repository:

```
Modules
| -- LocalMachine
     | -- LocalMachine.psd1
     | -- LocalMachine.psm1
```

Side-by-side versioning stores each version of a module in a versioned directory:

```
Modules
| -- LocalMachine
     | -- 1.0.0
          | -- LocalMachine.psd1
          | -- LocalMachine.psm1
```

This allows more than one version of a module to be installed. Commands such as `Import-Module` include parameters used to select a version.

This affects publishing a module in that a versioned directory is also permitted as the value for the `Path` parameter of `Publish-Module`. That is, if the project were laid out as the following illustrates:

```
ProjectRoot
| -- LocalMachine
     | -- 1.0.0
          | -- LocalMachine.psd1
          | -- LocalMachine.psm1
```

Then the following command would succeed if executed from the `ProjectRoot` directory:

```
$params = @{
    Path       = '.\LocalMachine\1.0.0'
    Repository = 'PSLocal'
}
Publish-Module @params
```

Embedding a version number as shown in the previous example will satisfy the path constraint for `Publish-Module`, but it is an unlikely directory structure to use when writing a module.

The `LocalMachine` module only includes three functions and a module manifest at this point. As the module grows in complexity, attempting to keep writing the module in a single file becomes more difficult.

Multi-file module layout

Multi-file layouts typically aim to split the content of a project up into smaller and easier to maintain files. The most common strategy is to create one file for each function, class, or enumeration.

Files are categorized and grouped together to make it easier to find parts of the module. For example, the `LocalMachine` module might split up as follows:

```
ProjectRoot
| -- LocalMachine
     | -- public
     |     | -- Get-ComputerDescription.ps1
     |     | -- Set-ComputerDescription.ps1
     |
     | -- private
     |    | -- GetRegistryParameter.ps1
     |    | -- TestComputerDescriptionValue.ps1
     |
     | -- LocalMachine.psd1
     | -- LocalMachine.psm1
```

When split like this, `LocalMachine.psm1` needs to change to load the content from those separate files.

Dot sourcing module content

Dot sourcing is used to load the content of a target file into the current scope. If used inside the `psm1` file, each individual file will be loaded into the same scope as the `psm1` file.

When the `psm1` file runs, the `$PSScriptRoot` variable (introduced in PowerShell 3) will be set to the directory the `psm1` file is saved in. This can be used to create an expression to find the files that make up the module content:

```
'public', 'private' |
    Resolve-Path -Path $PSScriptRoot -ChildPath { $_ } |
    Get-ChildItem -Recurse -File -Filter *.ps1
```

Note that this cannot be run outside of a script (or script module); the `$PSScriptRoot` variable will not exist.

> **Getting the module base**
>
> `$PSScriptRoot` is specific to the file it is used in. Therefore, if one of the dot sourced files from a sub-directory requires access to the module base directory, it is less useful.
>
> Instead, a property of the reserved variable, `$MyInvocation`, can be used from any script or function within a module to get the folder for the module manifest file:
>
> ```
> $MyInvocation.MyCommand.Module.ModuleBase
> ```

Each of the discovered files needs to be dot sourced. You can use the result of the preceding command in a loop or with `ForEach-Object`:

```
'public', 'private' |
    Resolve-Path -Path $PSScriptRoot -ChildPath { $_ } |
    Get-ChildItem -Recurse -File -Filter *.ps1 |
    ForEach-Object {
        . $_.FullName
    }
```

Explicitly naming the directories at the top level, that is, `public` and `private`, ensures that some control is maintained over the loading order of the parts of the module. For functions, this is unlikely to be required, but if classes are introduced, the order may become important.

This approach has a small risk in that it will load any other `ps1` files present in the module directory regardless of whether they belong to the module.

An alternative, but a higher maintenance approach is to name the files to import instead of allowing any file at all to load. For example:

```
$private = 'GetRegistryParameter'
foreach ($item in $private) {
    . '{0}\private\{1}.ps1' -f $PSScriptRoot, $item
}
$public = @(
    'Get-ComputerDescription'
)
foreach ($item in $public) {
    . '{0}\public\{1}.ps1' -f $PSScriptRoot, $item
}
Export-ModuleMember -Function $public
```

With this version, module content is loaded with an explicit name. Additional files that are erroneously placed in the module directory are not processed.

Dot sourcing module content is useful when a module is being developed. It allows a developer to realize the benefits of having module content split into separate files: content is easier to find, files are easier to read, and when source control is in use, changes are easier to review.

Conversely, modules split into lots of files are a little slower to load; potentially a lot slower if code signing is used. Code signing itself is beyond the scope of this chapter as few publicly available modules are signed. The topic is explored in the `about_Signing` help topic.

Merging module content

Merging module content using a build process of some kind offers the best of both worlds. The module is split into files, making it easy to write and edit; the module is merged into a single file, making it quick to load and easy to sign.

The `ModuleBuilder` module is capable of merging module content into a single file. The `ModuleBuilder` module requires a `build.psd1` file in the root of the module (adjacent to `LocalMachine.psd1`). The `build.psd1` file does not need to contain more than an empty Hashtable; it can be used to customize the merge process.

The following example instructs `Build-Module` to place merged content in a build directory and includes a versioned directory that might be used with `Publish-Module`:

```
@{
    ModuleManifest           = 'LocalMachine.psd1'
    OutputDirectory          = '../build'
    VersionedOutputDirectory = $true
}
```

With the file present, you can use the following command to merge the module content:

```
Install-Module ModuleBuilder -Scope CurrentUser
Build-Module -SourcePath .\LocalMachine
```

If the command is run from the same directory as `build.psd1`, the `SourcePath` argument can be omitted. For example:

```
Set-Location .\LocalMachine
Build-Module
```

The resulting module file is placed in the output directory under the project directory. The output path is configurable.

`ModuleBuilder` will generate a new module manifest based on the existing manifest. It will update `FunctionsToExport` based on the content of the public directory, and it will fill `AliasesToExport` if any of the commands use aliases.

As a module grows in complexity, it may be desirable to perform additional tasks during the build step. For example, you might add extra metadata to the module manifest, run tests, regenerate help files, or publish the module.

ModuleBuilder and DSC resources

`ModuleBuilder` will, by default, merge content from a subdirectory named `classes` into the generated root module. However, `ModuleBuilder` will not update the `DscResourcesToExport` property in the module manifest.

The `classes` directory in the module contains a version of the DSC class created in *Chapter 19, Classes and Enumerations*. The simplified version of the class is available on GitHub: https://github.com/PacktPublishing/Mastering-Windows-PowerShell-Scripting-Fourth-Edition/blob/master/Chapter20/LocalMachine/MultiFileWithModuleBuilder/LocalMachine/classes/ComputerDescription.ps1.

It is possible to dynamically update `DscResourcesToExport` but finding the resources in the module requires an advanced technique. The following snippet uses the **Abstract Syntax Tree** or **AST**, which is explored in more detail in *Chapter 21, Testing*.

The snippet is saved in a `build.ps1` file in the preceding repository in the `MultiFileWithModuleBuilder` directory, the project root for this module layout. The `build.ps1` file runs `Build-Module` and then updates the missing information in the module manifest based on the search below. The `Update-ModuleManifest` command from `PowerShellGet` is used to fill in the missing values in the manifest.

The snippet can be run from the `MultiFileWithModuleBuilder` directory:

```
using namespace System.Management.Automation.Language
# Find the root module
$rootModulePath = @{
    Path      = $pwd
    ChildPath = 'build\*\*\*.psm1'
}
$rootModule = Join-Path @rootModulePath | Resolve-Path
# These values do not need to be captured for this search process
$tokens = $errors = $null
$ast = [Parser]::ParseFile(
    $rootModule,
    [ref]$tokens,
    [ref]$errors
)
$dscResourcesToExport = $ast.FindAll({
    param ( $node )
    $node -is [TypeDefinitionAst] -and
    $node.IsClass -and
    $node.Attributes.TypeName.FullName -contains 'DscResource'
}, $true).Name
```

The preceding code reads the content of the `psm1` file generated by `ModuleBuilder`, parses that file, and searches for every PowerShell class that has the `DscResource` attribute. It outputs the names of the classes, which can then be used with the `Update-ModuleManifest` command:

```
$moduleManifestPath = @{
    Path      = $pwd
    ChildPath = 'build\*\*\*.psd1'
}
# Find the module manifest
$moduleManifest = Join-Path @moduleManifestPath |
    Get-Item |
    Where-Object { $_.BaseName -eq $_.Directory.Parent.Name }
$updateParams = @{
    Path                 = $moduleManifest
    DscResourcesToExport = $dscResourcesToExport
}
Update-ModuleManifest @updateParams
```

The technique above is complex but shows how PowerShell can be used to create a generic process that can be used to work on different kinds of modules without building a specialized process every time.

Module scope

Script modules loaded from `psm1` files have a shared scope that you can access using the `$Script:` scope modifier.

You can create variables in the module scope, which is the same as the script scope. Functions within the module may consume those variables. You can create such variables in the root module or when a command is run.

Helper functions can be created to provide obvious access to the variable content.

This approach is illustrated in the following example. This pattern can be used for modules that interact with services that require a connection or authentication. For example, a REST web service might require an explicit authentication step to acquire a time-limited token or key.

First, a function styled like the following `Connect-Service` function establishes a script-scoped variable. The connection can capture an authentication token rather than representing a truly connected service (for instance, a connection to an SQL server). The implementation of the command depends on the needs of the service and the module:

```
function Connect-Service {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory)]
        [String]$Server
    )
    $Script:connection = [PSCustomObject]@{
        Server     = $Server
        PSTypeName = 'ServiceConnectionInfo'
    }
}
```

You can create a command to allow either another command in the module or the end user to get the stored connection object:

```
function Get-ServiceConnection {
    [CmdletBinding()]
    param ( )
    if ($Script:connection) {
        $Script:connection
    } else {
        throw [InvalidOperationException]::new(
            'Not connected to the service'
        )
    }
}
```

Finally, the stored connection can be used by another function in the module to carry out the actions it needs:

```
function Get-ServiceObject {
    [CmdletBinding()]
    param (
        [PSTypeName('ServiceConnectionInfo')]
        $Connection = (Get-ServiceConnection)
    )
}
```

Defining the connection as a parameter allows the user of the function to provide an explicit connection if more than one connection is to be used. The `Connect-Service` function would have to be adjusted to accommodate such a scenario.

Accessing module scope

All functions and classes within a module can access script-scoped variables by using the scope modifier.

The script scope of a module can be accessed from outside the module to aid debugging by using the call operator.

The following snippet is used to demonstrate accessing module scope. The module consists of one public function, one private function, and a module-scoped variable.

The module is created in memory using the `New-Module` command, which avoids the need to create a file to demonstrate the feature. Note that the newly created module is piped to `Import-Module`, which ensures the module is accessible after import using `Get-Module`:

```
New-Module SomeService {
    function GetServiceConnection {
        [CmdletBinding()]
        param ( )
        $Script:connection
    }
    function Connect-Service {
        [CmdletBinding()]
        param (
            [String]$Name
        )
        $Script:connection = $Name
    }
    $Script:connection = 'DefaultConnection'
    Export-ModuleMember -Function Connect-Service
} | Import-Module
```

By default, only the `Connect-Service` function is available, as shown here:

```
PS> Get-Command -Module SomeService
CommandType     Name               Version    Source
-----------     ----               -------    ------
Function        Connect-Service    0.0        SomeService
```

The value of the script-scoped connection variable may be retrieved as follows. The module used in this manner must be imported prior to use:

```
PS> & (Get-Module SomeService) { $connection }
DefaultConnection
```

This technique can be used with any module, but it likely only makes sense when used with a script module (or a manifest module that includes script modules).

If the `Connect-Service` command is used, the value returned by the preceding command will change. The same approach may be used to interact with functions that are not normally exported by the module. For example, the `GetServiceConnection` function can be called:

```
& (Get-Module SomeService) { GetServiceConnection }
```

Finally, as the command is executing in the scope of the module, this technique may be applied to list the commands within the module, as follows:

```
PS> & (Get-Module SomeService) { Get-Command -Module SomeService }
CommandType     Name                    Version    Source
-----------     ----                    -------    ------
Function        Connect-Service         0.0        SomeService
Function        GetServiceConnection    0.0        SomeService
```

This technique is useful when debugging modules that heavily utilize module scope.

Module scope is also used when creating classes and enumerations within a module.

Modules, classes, and enumerations

PowerShell classes and enumerations are not exported from a module and are not made available to an end user when `Import-Module` is used, meaning they cannot be used outside of the module scope by default.

Classes and enumerations are only available outside of the module scope if a `using module` statement is used.

You can use classes as return values from functions within a module without changing scope. Enumeration values can be supplied as arguments for function parameters via the value name without needing direct access to the `enumeration` type. This is shown in the following example:

```
New-Module ModuleWithEnum {
    enum ParameterValues {
        ValueOne
        ValueTwo
    }
    function Get-Something {
        [CmdletBinding()]
        param (
            [ParameterValues]$Parameter
        )
    }
} | Import-Module
Get-Something -Parameter ValueOne
```

The preceding `Get-Something` function offers tab completion with names from the enumeration. A string, such as `ValueOne`, can refer to a value in the enumeration. The enumeration itself does not have to be made available to the global (or the caller) scope.

An instance of a PowerShell class can be returned as objects from functions within a module. This is shown in the following example:

```
New-Module ModuleWithClass {
    class ModuleClass {
        [string] $Property = 'value'
    }
    function Get-Something {
        [ModuleClass]::new()
    }
} | Import-Module
```

Running the `Get-Something` command will show that it returns an instance of the class:

```
PS> Get-Something
Property
--------
value
```

However, an instance of the class cannot be created in the global scope by default:

```
PS> [ModuleClass]::new()
InvalidOperation: Unable to find type [ModuleClass].
```

The `using` module is required to directly work with a class inside a module. Unfortunately, this is not possible with a module created by `New-Module`. The content below should be saved into a `ModuleWithClass.psm1` file so it can be imported:

```
@'
class ModuleClass {
    [string] $Property = 'value'
}
function Get-Something {
    [ModuleClass]::new()
}
'@ | Set-Content -Path ModuleWithClass.psm1
```

Once the file is created, the module can be imported with a `using module` statement.

```
using module .\ModuleWithClass.psm1
```

The `using module` statement can import either a module using a path or a module by name. Once the module has been imported this way, you can use the class:

```
PS> [ModuleClass]::new()
Property
--------
value
```

It is possible to work around this limitation by moving class definitions into `ScriptsToProcess`. This is explored in the next section.

Initializing and removing modules

The content of the root module executes every time a module is imported. You can use a root module file to perform initialization steps, for example, initializing a cache, importing static data, or setting a default configuration.

These steps are extra code that you must add to the root module at the beginning or end of the file.

In some cases, it is desirable to execute actions before a module is imported or when a module is removed.

You can use the `ScriptsToProcess` property in a module manifest to execute code in the caller scope before a module is imported.

The ScriptsToProcess property

You can use the `ScriptsToProcess` property in the module manifest to run scripts in the caller scope before importing the module. The caller is the entity importing the module; if that is an end user in the global scope, then `ScriptsToProcess` is executed in the global scope. If the module is imported by another script, `ScriptsToProcess` is executed in that script scope.

There are no limitations on what actions can be performed in `ScriptsToProcess`, but it is important to remember that it affects the global scope. Using the `Remove-Module` command will not remove content added to the caller scope.

The `ScriptsToProcess` property can be used, or abused, to make classes available to a module user without requiring the use of the `using` module.

Since `ScriptsToProcess` acts in the caller scope, this is not always ideal. The following snippet sets up a module with a class in a script that is executed in the caller scope to demonstrate this limitation:

```
@'
class ModuleClass {
    [string] $Property = 'value'
}
'@ | Set-Content -Path 'Script.ps1'
@'
function Get-Something {
    [ModuleClass]::new()
}
'@ | Set-Content -Path Module.psm1
$manifestParams = @{
    Path            = 'Module.psd1'
    RootModule      = 'Module.psm1'
    ScriptsToProcess = 'Script.ps1'
}
New-ModuleManifest @manifestParams
```

You can import this module in the current scope to use the class:

```
PS> Import-Module -Name .\Module.psd1
PS> [ModuleClass]::new()
Property
--------
value
```

PowerShell must be restarted to show the change in behavior when running the same command inside a function:

```
function Import-Something {
    Import-Module -Name .\Module.psd1 -Scope Global
}
```

Running `Import-Something` will, as the command says, import the module in the global scope. However, `ScriptsToProcess` still executes in the caller scope, that is, the scope of the function. Therefore, the class, exposed only via `ScriptsToProcess`, is still not available in the global scope, as the following shows:

```
PS> Import-Something
PS> Get-Module Module
ModuleType Version PreRelease Name   ExportedCommands
---------- ------- ---------- ----   ----------------
Script     0.0.1              Module Get-Something
PS> [ModuleClass]::new()
InvalidOperation: Unable to find type [ModuleClass].
```

`ScriptsToProcess` can be useful in the right circumstances, but it is not a fix for every problem with module content.

`Remove-Module` will attempt to remove module content from the session. If explicit actions are required when a module is removed, the `OnRemove` event can be used.

The OnRemove event

The `OnRemove` event is raised when you use `Remove-Module`. Any script block assigned to the event as a handler will be invoked. The event handler may be used to trigger a cleanup of the artifacts created by the module (if required).

The `ModuleInfo` object provides access to the `OnRemove` event handler via the `$executionContext` variable when it is used in the `psm1` file:

```
$executionContext.SessionState.Module.OnRemove
```

The following module creates a file named `log.txt` when the module is imported. An `OnRemove` handler is added to the module to attempt to release a lock created by the `StreamWriter` on the file when the module is removed:

```
@'
using namespace System.IO
$path = Join-Path $PSScriptRoot -ChildPath 'OnRemove.log'
$stream = [StreamWriter][File]::OpenWrite($path)
$stream.WriteLine('Initialising module')
$executionContext.SessionState.Module.OnRemove = {
    $stream.WriteLine('Closing log')
    $stream.Flush()
    $stream.Close()
}
'@ | Set-Content OnRemove.psm1
```

The file is created when the module is imported, and a `StreamWriter` is created:

```
PS> Import-Module -Name .\OnRemove.psm1
PS> Get-ChildItem -Path OnRemove.log
        Directory: C:\Workspace
Mode                LastWriteTime         Length Name
----                -------------         ------ ----
-a---        17/04/2021     10:19              0 OnRemove.log
```

Commands such as `Get-Content` cannot read the file; it is locked by the `StreamWriter`.

```
PS> Get-Content .\OnRemove.log
Get-Content: The process cannot access the file 'C:\Workspace\OnRemove.log' because it is being used
by another process.
```

Running `Remove-Module` triggers the `OnRemove` event. The `OnRemove` event writes one final line, flushes the buffer (`StreamWriter` does not immediately write to the file), and closes the stream.

Once the stream is closed, `Get-Content` is able to read the file content:

```
PS> Get-Content -Path OnRemove.log
Initialising module
Closing log
```

This event only runs when `Remove-Module` is used; it does not trigger if the PowerShell console is closed. In this case, the only difference is that the `Closing log` line is not written.

Summary

Modules are an important part of PowerShell as they encapsulate related functions (and other content). You can publish modules to public repositories such as the PowerShell Gallery, or any internal PowerShell repositories that may be in use.

Module layout in PowerShell is flexible beyond the creation of a `psm1` file (or a `dll` file in the case of a binary module).

You can optionally include a `psd1` file to describe information about the module, including versions, what should be made available to anyone importing the module, and other metadata such as project information.

Multi-file module layouts are common as they allow authors to manage content in separate files rather than a single monolithic root module. Some module authors choose to merge content into a single file as part of a publishing process, while others are happy to load individual files via the root module.

You can use module scope to store the information a module needs to work; for example, authentication tokens used with a remote service.

You can use PowerShell classes within a module and return instances of classes via functions to users of a module. Directly using classes within a module requires the use of the `using module` statement.

You can perform actions before importing a module via the `ScriptsToProcess` property in a module manifest and when removing a module using `Remove-Module` via the `OnRemove` event.

In the next chapter, *Chapter 21*, *Testing*, you'll explore static analysis and unit testing in PowerShell.

21

Testing

The goal of testing in PowerShell is to ensure that the code works as intended. Automatic testing ensures that this continues to be the case as code is changed over time.

Testing often begins before code is ready to execute. `PSScriptAnalyzer` can look at code and provide advice on possible best practices that help prevent common mistakes. `PSScriptAnalyzer` uses what is known as static analysis.

Unit testing, the testing of the smallest units of code, starts when the code is ready to execute. Tests can be created before the code when following practices such as **Test-Driven Development** (**TDD**). A unit test focuses on the smallest parts of a module, the functions and classes. A unit test strives to validate the inner workings of a unit of code, ensuring that conditions evaluate correctly, that it terminates or returns where it should, and so on.

Testing might extend into systems and acceptance testing, although this often requires a test environment to act against. Acceptance testing may include black-box testing, used to verify that a command accepts known parameters and generates an expected set of results. Black-box testing, as the name suggests, does not concern itself with understanding how a block of code arrives at a result.

This chapter covers the following topics:

- Static analysis
- Testing with Pester

Before beginning with the main topics of the chapter, there are some technical requirements to consider.

Technical requirements

The following modules are used in this chapter:

- PSScriptAnalyzer 1.19.1
- Pester 5.1.1
- ShowPSAst 1.0

Static analysis

Static analysis is the process of evaluating code without executing it. `PSScriptAnalyzer` uses static analysis.

In PowerShell, static analysis most often makes use of an **Abstract Syntax Tree** (**AST**): a tree-like representation of a piece of code. In PowerShell, an element of a script is represented by a node in the syntax tree. AST was introduced with PowerShell 3.

The largest elements represent the script itself, the root of the tree in effect. Each element added to the script is represented by a child node. For example, the parameter block is described by a `ParamBlockAst` object, an individual parameter by a `ParameterAst`, and so on.

Evaluating elements of the AST is the basis for many of the rules implemented in `PSScriptAnalyzer`.

PSScriptAnalyzer

The `PSScriptAnalyzer` module is used to run a series of rules against a file or string containing a script. You can install the tool can using the following code:

```
Install-Module PSScriptAnalyzer
```

You can use `PSScriptAnalyzer` to inspect a script with the `Invoke-ScriptAnalzyer` command. For example, the tool will raise one error and one warning for the following script:

```
@'
[CmdletBinding()]
param (
    [Parameter(Mandatory)]
    [String]$Password
)
$credential = [PSCredential]::new(
    '.\user',
    ($Password | ConvertTo-SecureString -AsPlainText -Force)
)
$credential.GetNetworkCredential().Password
'@ | Set-Content Show-Password.ps1
```

When `Invoke-ScriptAnalyzer` is run on the file, two rule violations are shown, one for the use of `ConvertTo-SecureString`, and one for the `$Password` parameter using plain text:

```
PS> Invoke-ScriptAnalyzer .\Show-Password.ps1 | Format-List
RuleName : PSAvoidUsingConvertToSecureStringWithPlainText
Severity : Error
Line     : 9
Column   : 18
Message  : File 'Show-Password.ps1' uses ConvertTo-SecureString
           with plaintext. This will expose secure information.
           Encrypted standard strings should be used instead.
RuleName : PSAvoidUsingPlainTextForPassword
Severity : Warning
Line     : 3
Column   : 5
Message  : Parameter '$Password' should use SecureString,
           otherwise this will expose sensitive information. See
           ConvertTo-SecureString for more information.
```

This is one of many best-practice style rules that you can use to test a script.

Configurable rules

`PSScriptAnalyzer` includes 64 default rules. Most of these rules are automatically evaluated when a script is analyzed. Several rules require configuration before they can be used; these are not enabled by default.

The following command shows the rules that must be explicitly configured before they can be applied:

```
Get-ScriptAnalyzerRule | Where-Object {
    $_.ImplementingType.BaseType.Name -eq 'ConfigurableRule'
}
```

Conversely, the rules that are evaluated by default (without extra configuration) are shown with the following command:

```
Get-ScriptAnalyzerRule | Where-Object {
    $_.ImplementingType.BaseType.Name -ne 'ConfigurableRule'
}
```

Configurable rules may be configured using either a settings file or a Hashtable that describes the configuration for a rule. The following example shows how to use the `PSUseCorrectCasing` rule against a script read from a string:

```
$params = @{
    ScriptDefinition = 'get-process'
    Settings = @{
        Rules = @{
            PSUseCorrectCasing = @{
                Enable = $true
            }
        }
    }
}
Invoke-ScriptAnalyzer @params
```

Once the configuration for the rule is added, the rule will execute according to the settings for that rule. The settings for the rule are documented in the `PSScriptAnalyzer` repository. The document for `PSUseCorrectCasing` is: https://github.com/PowerShell/PSScriptAnalyzer/blob/master/RuleDocumentation/UseCorrectCasing.md .

Note that the rule documentation omits the `PS` prefix on the rule name. Several other rules, such as `PlaceOpenBrace` require configuration in a similar manner.

`PSScriptAnalyzer` includes several built-in settings files, which will tab complete when using the `Settings` parameter. For example:

```
Invoke-ScriptAnalyzer .\Show-Password.ps1 -Settings CodeFormatting
```

The settings used by each are not documented in Help for the module, but the content of each file can be viewed in the module directory. You can either use these files or as an example to build a customized settings file.

You can view the settings files shipped with `PSScriptAnalyzer` on GitHub: https://github.com/PowerShell/PSScriptAnalyzer/tree/master/Engine/Settings.

It is sometimes hard to meet the requirements of all rules in `PSScriptAnalyzer`. Rules can be suppressed globally in the settings file, or rules can be suppressed in code.

Suppressing rules

It is rarely realistic to expect any significant piece of code to pass all the tests that `PSScriptAnalyzer` will throw at it.

Individual tests can be suppressed at the function, script, or class level. The following demonstrative function creates a `PSCustomObject`:

```
@'
function New-Message {
    [CmdletBinding()]
    param (
        $Message
    )
    [PSCustomObject]@{
        Name  = 1
        Value = $Message
    }
}
'@ | Set-Content New-Message.ps1
```

Running `PSScriptAnalyzer` against a file containing the function will show the following warning:

```
PS> Invoke-ScriptAnalyzer -Path .\New-Message.ps1 | Format-List
RuleName : PSUseShouldProcessForStateChangingFunctions
Severity : Warning
Line     : 1
Column   : 10
Message  : Function 'New-Message' has verb that could change
           system state. Therefore, the function has to support
           'ShouldProcess'.
```

Given that this function creates a new object in the memory, and does not change the system state, the message might be suppressed. This is achieved by adding a `SuppressMessage` attribute before a `param` block:

```
function New-Message {
    [Diagnostics.CodeAnalysis.SuppressMessage(
        'PSUseShouldProcessForStateChangingFunctions',
        ''
    )]
    [CmdletBinding()]
    param (
        $Message
    )

    [PSCustomObject]@{
        Name  = 1
        Value = $Message
    }
}
```

`PSScriptAnalyzer` leverages an existing class from .NET to express suppressed rules. Visual Studio Code will offer to create an attribute when you start to type "suppress." The second argument, set to an empty string in the preceding example, is required but will be empty in most cases.

Once added, `Invoke-ScriptAnalyzer` will cease to warn of the rule failure for this function only.

AST is the basis for the majority of the rules used by `PSScriptAnalyzer`.

Using AST

The AST in PowerShell is available for any script block; an example is as follows:

```
PS> { Write-Host 'content' }.Ast
Attributes         : {}
UsingStatements    : {}
ParamBlock         :
BeginBlock         :
ProcessBlock       :
EndBlock           : Write-Host 'content'
DynamicParamBlock  :
ScriptRequirements :
Extent             : { Write-Host 'content' }
Parent             : { Write-Host 'content' }
```

The object returned describes each of the elements of the script (or script block in this case). It shows that the command in the script block is in the end block, the default block.

The script block that defines a function can be retrieved via `Get-Command`:

```
function Write-Content { Write-Host 'content' }
(Get-Command Write-Content).ScriptBlock
```

Or the script block defining a function can be retrieved using `Get-Item`:

```
function Write-Content { Write-Host 'content' }
(Get-Item function:\Write-Content).ScriptBlock
```

The preceding approaches have one thing in common, PowerShell is immediately parsing the script and will stop if there are any parser errors. The impact of this can be seen if an error is inserted into a script block; the syntax tree will not be accessible:

```
PS> {
    Write-Host
    --String--
}
ParserError:
Line |
   3 |      --String--
     |        ~
     | Missing expression after unary operator '--'.
```

To allow access to the AST, regardless of errors in the script, you can use the `Parser` class to read content either from a file or from a string.

The `Parser` class is accessed under the `System.Management.Automation.Language` namespace. The following example uses the `ParseInput` method to read PowerShell content from a string:

```
using namespace System.Management.Automation.Language
$script = @'
Write-Host
--String--
'@
$ast = [Parser]::ParseInput($script,  [ref]$null, [ref]$null)
```

The `ParseFile` method can be used in place of `ParseInput`. The same arguments are used but the string containing the script can be replaced for a path to a file (a full path, not a relative path).

Two of the method arguments to the `ParseInput` method in the previous example are set as references to `$null`. This essentially means they are ignored at this point. Ordinarily, the first would be used to fill an existing array of tokens, the second an array of errors. Tokens are explored in more detail later in this section.

The errors array reference can be used to capture parse-time errors, such as the error shown when attempting to create the script block.

```
using namespace System.Management.Automation.Language
$errors = $tokens = @()
$script = @'
Write-Host
--String--
'@
$ast = [Parser]::ParseInput($script,  [ref]$tokens, [ref]$errors)
```

You can view the content of the array after the `ParseInput` method has completed:

```
PS> $errors | Format-List
Extent          :
ErrorId         : MissingExpressionAfterOperator
Message         : Missing expression after unary operator '--'.
IncompleteInput : False
Extent          : String--
ErrorId         : UnexpectedToken
Message         : Unexpected token 'String--' in expression or statement.
IncompleteInput : False
```

The original script block only showed one error, but parsing stopped at the first error. This approach shows all syntax errors. If attempting to fix such errors, a top-down approach is required; one syntax error can easily cause another.

Returning to the AST object, the object represents a tree, therefore it is possible to work down through the list of properties getting to more specific elements of the script. Each element has a different type. The following example includes `ScriptBlockAst`, `StatementAst`, `NamedBlockAst`, `PipelineAst`, and `CommandAst` (and more as the more detailed elements of the script are explored).

The following example gets the `CommandAst` for the `Get-Process` command. That is the part of the script that represents just the `Get-Process -ID $PID`:

```
$ast = { Get-Process -ID $PID | Select-Object Name, Path }.Ast
$ast.EndBlock.Statements[0].PipelineElements[0]
```

All named blocks, such as the `EndBlock` here, can contain zero or more statements. Each statement can contain one or more pipeline elements. The `Select-Object` command in this example is in index 1 of the `PipelineElements` property.

You can use a module called `ShowPSAst` to visualize the tree; the module uses Windows Forms to draw a GUI and is therefore only compatible with Windows systems.

Visualizing the AST

The `ShowPSAst` module, available in the PowerShell Gallery, may be used to visualize the AST tree. Install the module with:

```
Install-Module ShowPSAst -Scope CurrentUser
```

Once it's installed, you can use the `Show-Ast` command on a string, a function, a module, a script block, and so on. Running the following command will show the AST tree in an Ast Explorer window.

```
Show-Ast 'Get-Process -ID $PID | Select-Object Name, Path'
```

Figure 21.1 shows the explorer window:

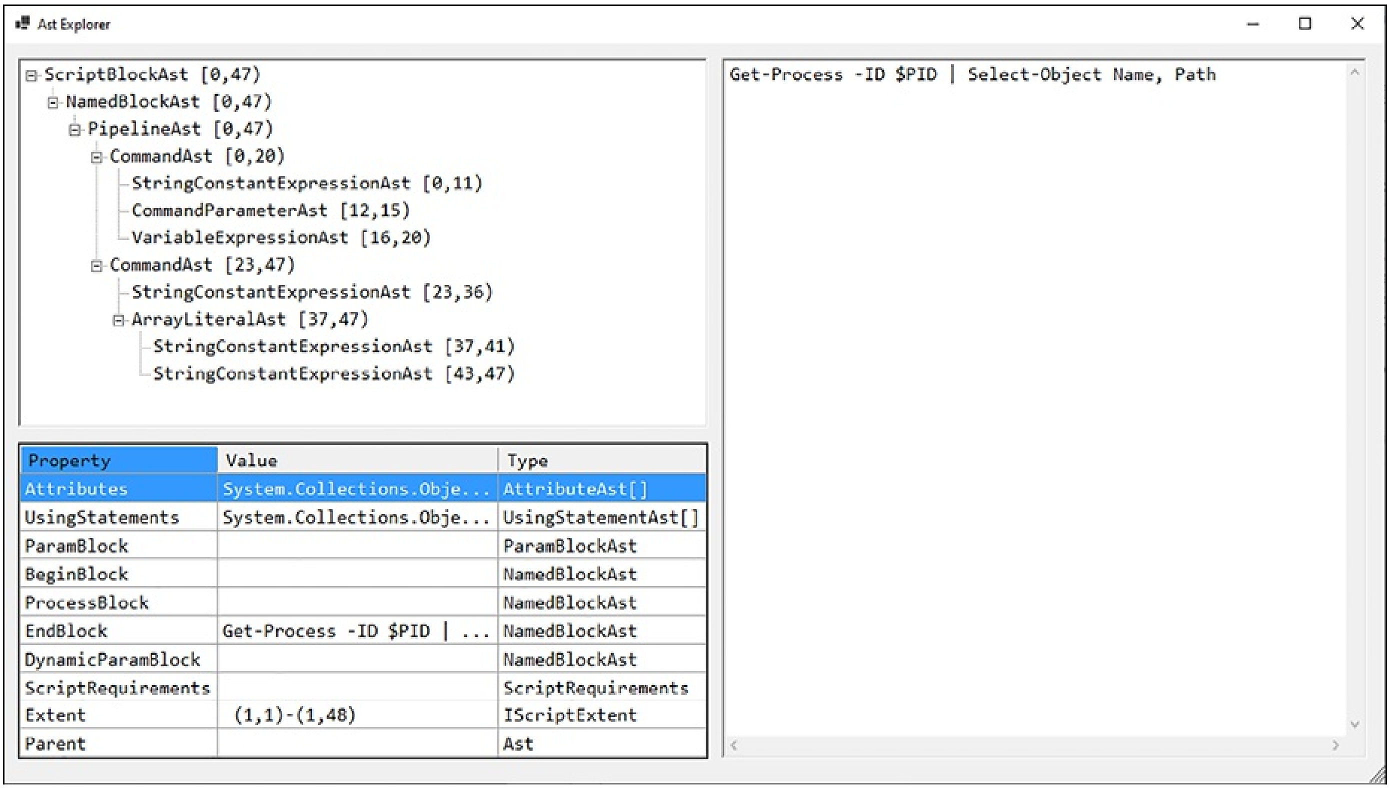


Figure 21.1: The Ast Explorer window

This tiny script with just one short line of code has 12 separate AST nodes in the tree. Attempting to access individual elements of

a script by expanding each property and array index quickly becomes impractical in large scripts.

It is possible to search the AST tree using `Find` and `FindAll` methods on any AST node.

Searching the AST

Searches against the AST can use the `Find`, which finds the first match only, and `FindAll` methods of any AST node. The methods find descendant nodes of the current node. Therefore, a search on a `PipelineAst` instance will only find results beneath that node.

An earlier example found the `CommandAst` for `Get-Process` by expanding properties in the AST:

```
$ast = { Get-Process -ID $PID | Select-Object Name, Path }.Ast
$ast.EndBlock.Statements[0].PipelineElements[0]
```

This can be rewritten to use the `Find` method instead. The key to the `Find` method is a predicate. The predicate is a script block that returns true or false. Each node is tested against the predicate and returned if the result is true.

The simplest predicate is therefore as follows:

```
$predicate = { $true }
```

When used with the `Find` method, the first matching node is returned. This will be the `ScriptBlockAst`, the top-most node in the tree. The second argument states whether the `Find` (or `FindAll`) method should search nested script blocks. More on this shortly:

```
using namespace System.Management.Automation.Language
$ast = { Get-Process -ID $PID | Select-Object Name, Path }.Ast
$predicate = { $true }
```

`Find` will show the `ScriptBlockAst` object and is therefore equal to the content of the `$ast` variable:

```
PS> $ast.Find($predicate, $true)
Attributes         : {}
UsingStatements    : {}
ParamBlock         :
BeginBlock         :
ProcessBlock       :
EndBlock           : Get-Process -ID $PID | Select-Object Name, Path
DynamicParamBlock  :
ScriptRequirements :
Extent             : { Get-Process -ID $PID | Select-Object Name,
                     Path }
Parent             : { Get-Process -ID $PID | Select-Object Name,
                     Path }
```

Finding the node for the `Get-Process` command requires a more complex predicate. Each node in the AST is passed as an argument into the predicate. This can either be accessed using `$args[0]` or by defining a parameter to accept that value (as the following code shows). The AST type required is `CommandAst`. `CommandAst` has a `GetCommandName` method, which can be used to separate the command name from the arguments. Here is the updated predicate:

```
using namespace System.Management.Automation.Language
$ast = { Get-Process -ID $PID | Select-Object Name, Path }.Ast
$predicate = {
    param ( $node )
    $node -is [CommandAst] -and
    $node.GetCommandName() -eq 'Get-Process'
}
```

This time the result of the search is the node describing `Get-Process`:

```
PS> $ast.Find($predicate, $true)
CommandElements    : {Get-Process, ID, $PID}
InvocationOperator : Unknown
DefiningKeyword    :
Redirections       : {}
Extent             : Get-Process -ID $PID
Parent             : Get-Process -ID $PID | Select-Object Name, Path
```

As shown in the preceding example, each AST node returns an `Extent` property. The `Extent` property describes information about the position of a node within the larger script, such as where it begins and ends.

```
PS> $ast.Find($predicate, $true).Extent
File                :
StartScriptPosition : System.Management.Automation.Language.Int...
EndScriptPosition   : System.Management.Automation.Language.Int...
StartLineNumber     : 1
StartColumnNumber   : 10
EndLineNumber       : 1
EndColumnNumber     : 30
Text                : Get-Process -ID $PID
StartOffset         : 9
EndOffset           : 29
```

Line and column numbers may vary depending on the source script.

This information can potentially be used to selectively edit a script if required. This technique is used by several commands in the `PSKoans` module (https://github.com/vexx32/PSKoans) to replace or update content in existing scripts.

As mentioned at the start of this section, searches like this are the basis for many of the rules in `PSScriptAnalyzer`. `PSScriptAnalyzer` supports a second type of rule, a rule based on tokens within a script.

Tokenizer

In addition to the AST, PowerShell can also convert a script into a series of tokens, each representing an element of a script with no hierarchy.

One of the advantages of the tokenizer is that it will return tokens representing comments, whereas the AST ignores comments entirely:

```
using namespace System.Management.Automation.Language
$errors = $tokens = @()
$script = @'
# A short script
Write-Host 'Hello world'
'@
$ast = [Parser]::ParseInput($script,  [ref]$tokens, [ref]$errors)
```

Once executed, the tokens that make up the script can be examined. The first two tokens are shown here:

```
PS> $tokens | Select-Object -First 2
Text       : # A short script
TokenFlags : ParseModeInvariant
Kind       : Comment
HasError   : False
Extent     : # A short script
Text       :
TokenFlags : ParseModeInvariant
Kind       : NewLine
HasError   : False
Extent     :
```

Tokens are less useful than the AST when it comes to defining rules. The lack of context makes it more difficult to relate one token to another beyond the order in the array. Tokens might be used in a rule to validate the content of comments if necessary.

The AST and tokens are used by `PSScriptAnalyzer` to implement rules.

Custom script analyzer rules

`PSScriptAnalyzer` allows custom rules to be defined and used. Custom rules might be used to test for personal or organization-specific conventions when striving for a consistent style; such conventions may not necessarily be widely adopted best practices, but instead locally established best practices.

Script analyzer rules must be defined in a module `psm1` file. The path to the module file may be passed in by using the `CustomRulePath` parameter or may be defined in a script analyzer configuration file.

Creating a custom rule

A script analyzer rule is a function within a module. The `PSScriptAnalyzer` module allows rules to be written to evaluate AST nodes or tokens.

The name of the function is arbitrary. The community examples use the verb `measure`; however, the use of this verb is not mandatory and does not affect discovery. The community example is linked here for reference: https://github.com/PowerShell/PSScriptAnalyzer/blob/master/ScriptRuleDocumentation.md.

The following examples use a much more lightweight format. This does not sacrifice functionality.

The script analyzer engine examines each function in the custom rule module, looking for parameters with a particular naming style. If a parameter is found, the function is deemed to be a rule.

If a rule is expected to act based on an AST node, the first parameter name must end with `ast`. The parameter must use one of the AST types, such as `System.Management.Automation.Language.ScriptBlockAst`.

If a rule is expected to act based on a token, the first parameter name must end with `token` and must accept an array of tokens.

AST-based rules

Script analyzer rules are often simple; it is not always necessary for a rule to perform complex AST searches.

The following example evaluates the named blocks `dynamicparam`, `begin`, `process`, and `end`. If one of the blocks is declared in a function, script, or script block, and it is empty, the rule will respond.

The rule only accepts `NamedBlockAst` nodes, the smallest scope for the rule to effectively evaluate the script. The script analyzer only passes nodes of that type to the rule, and therefore, the rule itself does not have to worry about handling other node types or performing searches itself.

The rule simply looks to see if the number of statements in the block is 0. If it is 0, then the rule triggers.

The following rule is expected to be placed in a `psm1` file. For the sake of this example, that file can be named `CustomRules.psm1`:

```
using namespace Microsoft.Windows.PowerShell.ScriptAnalyzer.Generic
using namespace System.Management.Automation.Language
function PSAvoidEmptyNamedBlocks {
    [CmdletBinding()]
    param (
        [NamedBlockAst]$ast
    )
    if ($ast.Statements.Count -eq 0) {
        [DiagnosticRecord]@{
            Message  = 'Empty {0} block.' -f $ast.BlockKind
            Extent   = $ast.Extent
            RuleName = $myinvocation.MyCommand.Name
            Severity = 'Warning'
        }
    }
}
```

The rule returns `DiagnosticRecord` when it is triggered. The record is returned by the script analyzer provided the rule is not suppressed. The next command shows the rule in action:

```
@'
[CmdletBinding()]
param ( )
begin { }
process { }
end {
    Write-Host 'Hello world'
}
'@ | Set-Content script.ps1
$params = @{
    Path           = 'script.ps1'
    CustomRulePath = '.\CustomRules.psm1'
}
Invoke-ScriptAnalyzer @params
```

The output from the command flags the `begin` and `process` blocks as they are empty:

```
PS> Invoke-ScriptAnalyzer @params
RuleName                 Severity ScriptName Line Message
--------                 -------- ---------- ---- -------
PSAvoidEmptyNamedBlocks Warning  script.ps1 4    Empty Begin ...
PSAvoidEmptyNamedBlocks Warning  script.ps1 5    Empty Process...
```

Token-based rules are written in a similar manner.

Token-based rules

Rules based on tokens evaluate an array of tokens. The following example looks for empty single-line comments in a block of code. Comments are not a part of the syntax tree, so using tokens is the only option. This new rule can be added to the `CustomRules.psm1` file created in the previous section:

```
using namespace Microsoft.Windows.PowerShell.ScriptAnalyzer.Generic
using namespace System.Management.Automation.Language
function PSAvoidEmptyComments {
    [CmdletBinding()]
    param (
        [Token[]]$token
    )
    $ruleName = $myinvocation.MyCommand.Name
    $token.Where{
        $_.Kind -eq 'Comment' -and
        $_.Text.Trim() -eq '#'
    }.ForEach{
        [DiagnosticRecord]@{
            Message  = 'Empty comment.'
            Extent   = $_.Extent
            RuleName = $ruleName
            Severity = 'Information'
        }
    }
}
```

As the name suggests, the rule will trigger when it encounters an empty line comment. This is demonstrated by the following example:

```
@'
[CmdletBinding()]
param ( )
#
# Comment
Write-Host 'Hello world'
'@ | Set-Content script.ps1
```

The output from `Invoke-ScriptAnalyzer` shows the line that failed:

```
PS> $params = @{
>>     Path           = 'script.ps1'
>>     CustomRulePath = '.\CustomRules.psm1'
>> }
PS> Invoke-ScriptAnalyzer @params
RuleName             Severity     ScriptName Line  Message
--------             --------     ---------- ----  -------
PSAvoidEmptyComments Information  script.ps1 4     Empty comment.
```

More custom rules

For more examples of custom rules, please see:

https://github.com/indented-automation/Indented.ScriptAnalyzerRules

`PSScriptAnalyzer` is a fantastic tool that can attempt to enforce a specific style or help fix common problems.

Testing with Pester

Pester is a framework for executing tests. It includes tools to define and execute test cases against anything that can be written in PowerShell.

This chapter focuses on Pester 5, the latest major release. Pester 5 is not installed by default; Windows ships with Pester 3.4. This pre-installed version can be ignored:

```
Install-Module Pester -Force -SkipPublisherCheck
```

The `-SkipPublisherCheck` parameter is required as Pester has changed maintainer since the version shipped with Windows was released. The certificate issued to the pre-installed version differs from the certificate issued to the current version.

You can use Pester to write tests for code and systems and everything in between. Pester is implemented as what is known as a **Domain-Specific Language**. It has specific functions that are implemented to behave like language keywords. For example, `function` is a language-specific keyword. Pester tests are written using PowerShell, but for the most part, the language, the keywords, and so on, are defined by and specific to Pester.

The following example creates a test that asserts PowerShell 7 or greater should be in use. If the test runs in Windows PowerShell, the test will fail, and the results of that failure will be displayed to someone running the test. Pester tests must be saved in a file in Pester 5 so the following snippet saves content to a file before running `Invoke-Pester`:

```
@'
Describe 'PS developer workstation' {
    It 'PowerShell 7 is installed' {
        $PSVersionTable.PSVersion |
            Should -BeGreaterOrEqual 7.0.0
    }
}
'@ | Set-Content workstation.tests.ps1
Invoke-Pester -Path workstation.tests.ps1
```

The outcome of running the test is displayed in the console, although the results of the test are summarized by default. Times to perform discovery and execute tests may vary:

```
PS> Invoke-Pester -Path workstation.tests.ps1
Starting discovery in 1 files.
Discovery finished in 5ms.
[+] C:\workspace\workstation.tests.ps1 91ms (2ms|86ms)
Tests completed in 93ms
Tests Passed: 1, Failed: 0, Skipped: 0 NotRun: 0
```

Setting the `-Output` parameter to `Detailed` will show the results of each of the tests performed in the script. This test script only has one test. The difference is relatively small:

```
PS> Invoke-Pester -Path workstation.tests.ps1 -Output Detailed
Starting discovery in 1 files.
Discovering in C:\workspace\workstation.tests.ps1.
Found 1 tests. 9ms
Discovery finished in 13ms.
Running tests from 'C:\workspace\workstation.tests.ps1'
Describing PS developer workstation
  [+]PowerShell 7 is installed 4ms (1ms|3ms)
Tests completed in 104ms
Tests Passed: 1, Failed: 0, Skipped: 0 NotRun: 0
```

Tests must be saved to a file

This section focuses on test file content and not on the process of saving that content to a file. Test content should be saved to a file and run in the same manner as the preceding examples.

Note that the `Invoke-Pester` command specifically looks for `.tests.ps1` in file names.

The previous example uses three of the major keywords used in Pester:

- `Describe` – Groups tests for a particular subject together
- `It` – Defines a single test that should be executed

- `Should` – Asserts what the value or result of an expression should be

The condition used with the `Should` keyword simply states that the major version number should be 7 or greater.

Testing methodologies

Testing is a complex topic; it encompasses a wide range of different methodologies and concepts. The majority of these are beyond the scope of this chapter. Further reading is available on sites such as Wikipedia: https://en.wikipedia.org/wiki/Software_testing.

Two methodologies are of interest in this chapter. They are:

- Acceptance testing
- Unit testing

Acceptance testing is used to validate that the subject of the tests conforms to a pre-defined state. The test for the version of PowerShell at the start of this section might be part of an acceptance test for a developer workstation.

Acceptance testing in relation to PowerShell development strives to test the outcome of actions performed by a command (or script) without having any knowledge of how that script works. Acceptance testing is, therefore, a form of black-box testing and requires a system that code can be run against.

Unit testing aims to test the smallest units of code and is a form of white-box testing. The author of a unit test must be familiar with the inner workings of the subject of the tests.

Unit testing is most relevant in PowerShell when testing that the components of a module behave as they are expected to behave; that the different paths through a function, based on `if` statements and loops, are used correctly. Unit testing does not require a live service to act on. External calls are mocked, a fake response is returned. Mocking is explored later in this chapter.

The advantage of putting tests in code is that they can be run whenever the state of the subject changes. It is possible to continue to prove that the subject of a set of tests is working as expected.

One of the most challenging aspects of any testing process is figuring out what should be tested.

What to test

When testing systems, or performing acceptance testing, the following are rough examples of things that might be tested:

- Installed software packages
- File system paths or environment variables
- Application or service configuration
- Responses from remote systems the subject is expected to interact with
- Network access and network configuration.

When testing a module, or performing unit testing, consider testing the following:

- Parameters (and parameter binding)
- Any complex conditions and branches (conditional statements and loops) in the code
- Acceptance of different input or expected values, including complex parameter validation
- Exit conditions, especially raised errors or exceptions

When writing a unit test, resist the temptation to test other functions or commands called by the unit of code. A unit test is not responsible for making sure that every command that it calls works.

How extensive tests should be is debatable. Enough to ensure the functionality of a given script or module is perhaps the only real definition.

Code coverage is one of the measures that is often used. It is the percentage of code that is visited when executing a set of tests. Pester is capable of measuring code coverage. The details of this are shown later in this section. However, while this is an interesting indicator, it does not prove that code has been effectively tested.

Perhaps the most important keywords in Pester are `Describe`, `It`, and `Should`. They are the backbone of any set of tests.

Describing tests

Pester includes keywords that are used to enclose and group tests together. This section explores the keywords that are used to enclose tests. They are:

- `Describe`
- `Context`
- `It`

The `Describe` and `Context` keywords are both used to enclose or group together sets of tests. The tests themselves are defined within an `It` statement.

All test documents will include the `Describe` keyword and one or more `It` statements.

About the Describe and Context keywords

`Describe` is the top-most keyword used in a test document. It most often describes the subject of the tests.

`Context` is essentially the same as `Describe`. It has the same capabilities and will contain one or more `It` statements. `Context` is typically used under `Describe` to group together small sets of tests, typically where the tests have a similar purpose or require similar start conditions.

A test document might have a broadly defined subject, and several more specifically defined components. For example, a set of tests might be developed to describe the expected state of a developer workstation. The tests are broken down into more detailed sub-sections:

```
Describe 'PS developer workstation' {
    Context 'PowerShell' {
    }
    Context 'Packages' {
    }
}
```

The use of `Context` will become clear as tests grow in complexity and unit tests against PowerShell code are introduced later in this chapter.

`Describe`, or each `Context`, can include one or more `It` blocks, which describe the expected outcome.

About the It keyword

The `It` keyword is used to define a single test. The test title should describe the purpose and potentially the expected outcome of the test:

```
Describe 'PS developer workstation' {
    Context 'PowerShell' {
        It 'PowerShell 7 is installed' {
        }
    }
    Context 'Packages' {
        It 'git is installed' {
        }
        It 'Terraform is installed' {
        }
    }
}
```

The `It` keyword will contain one or more assertions using the `Should` keyword.

Should and assertions

The `Should` keyword is used to assert the state of the thing it is testing.

`Should` has 25 different parameter sets, one for each of the assertions it supports. The different assertions are documented in the Pester wiki along with an example: https://pester.dev/docs/assertions/assertions.

The example used at the start of this section uses one of the possible assertions, the `-BeGreaterOrEqual` assertion. Assertions have unsurprising options for the most part. If testing a Boolean value, `-BeTrue` or `-BeFalse` are appropriate.

Comparisons are achieved using `-Be`, `-BeLessThan`, `-BeLessOrEqual`, `-BeGreaterThan`, `-BeGreaterOrEqual`, and so on.

The first of the tests, the test for the installation of PowerShell 7, can be changed to allow it to run in Windows PowerShell as well. The point is to prove the system is in the expected state, not that the current runtime is PowerShell 7. The second `Context` is temporarily removed to focus on this one assertion.

```
Describe 'PS developer workstation' {
    Context 'PowerShell' {
        It 'PowerShell 7 is installed' {
            Get-Command pwsh -ErrorAction SilentlyContinue |
                ForEach-Object Version |
                Should -BeGreaterOrEqual '7.0.0'
        }
    }
}
```

Testing for errors is perhaps one of the most complex assertions and benefits from more extensive exploration. You can use a `Should -Throw` assertion to test whether a specific error is raised (or not) when running a command.

Testing for errors

The `-Throw` assertion is used to test whether a block of code throws a terminating error such as one raised when `ErrorAction` is set to `Stop` or when the `throw` keyword is used.

The assertion can be used for several of the tests above, but for the sake of variety, it is only used to test for the installation of Chocolatey, a package manager for Windows:

```
Describe 'PS developer workstation' {
    Context 'PowerShell' {
        It 'PowerShell 7 is installed' {
            Get-Command pwsh -ErrorAction SilentlyContinue |
                ForEach-Object Version |
                Should -BeGreaterOrEqual '7.0.0'
        }
    }
    Context 'Packages' {
        It 'Chocolatey is installed' {
            { Get-Command choco -ErrorAction Stop } |
                Should -Not -Throw
        }
    }
}
```

Notice how the expression being tested is defined as a script block, and that the script block is piped to the `Should` keyword.

The assertion used in the preceding example expects there to be no error. There is therefore no need to test anything else about the error.

If an error is expected to be thrown, then further tests might be beneficial. For example, attempting to divide 1 by 0 will raise an error:

```
Describe Division {
    It 'Throws an error when 1 is divided by 0' {
        { 1/0 } | Should -Throw
    }
}
```

This type of test is not specific; it does not differentiate between the actual problem and any other error that might occur. Changing the assertion, as the following shows, will still correctly identify that an error is thrown, but the error is no longer consistent with the descriptive name for the `It` statement:

```
Describe Division {
    It 'Throws an error when 1 is divided by 0' {
        { throw } | Should -Throw
    }
}
```

If this is saved to a `division.tests.ps1` file, it can be run to show that the test passes:

```
PS> Invoke-Pester -Path .\division.tests.ps1
Starting discovery in 1 files.
Discovery finished in 5ms.
[+] C:\workspace\division.tests.ps1 100ms (4ms|93ms)
Tests completed in 102ms
Tests Passed: 1, Failed: 0, Skipped: 0 NotRun: 0
```

Adding the `-ExpectedMessage` parameter is one way to tackle this. Testing for a specific message will greatly improve the accuracy of the test:

```
Describe Division {
    It 'Throws an error when 1 is divided by 0' {
        { 1/0 } | Should -Throw -ExpectedMessage 'Attempted to divide by zero.'
    }
}
```

For the preceding exception, testing the message is potentially as good as it gets. However, since error messages are often written in a user's language, testing the message is a weak test as it demands the tests are run in a specific culture.

The `-Throw` assertion allows both the error type and the fully qualified error ID to be tested instead. These are far more robust if the expression raising the error reveals them. The following example tests the fully qualified error ID:

```
Describe ErrorID {
    It 'Raises an error with a fully-qualified error ID' {
        { Write-Error error -ErrorID SomeErrorID -ErrorAction Stop } |
            Should -Throw -ErrorId SomeErrorID
    }
}
```

This type of testing is far more accurate, it may be possible to attribute the `ErrorID` to a single statement in the code being tested rather than testing for any error anywhere.

It is frequently necessary to perform setup actions prior to executing tests. Pester includes several named blocks for this purpose.

Iteration with Pester

It is often desirable to repeat the same test or tests for a different subject. Pester offers two different styles of iteration:

- The `It` keyword has the `TestCases` parameter
- The `Describe` and `Context` keywords have the `ForEach` parameter

The `TestCases` parameter for an `It` statement allows a single test to be executed against a set of predefined cases.

Using the TestCases parameter

The packages context of the "PS developer workstation" acceptance tests are a good candidate for test cases. A few packages were listed in the example context. The following tests assert that each of these should have been installed using Chocolatey, a package manager for Windows that can be downloaded from https://chocolatey.org.

With Chocolatey, the installation of a package can be tested using the following command:

```
choco list -e terraform -l -r
```

If the exact package name (`-e`) is installed locally (`-l`) it will be included in the output from the command. The `-r` parameter is used to limit output to essential information only, in this case just the package name and version.

The version of the installed package is not relevant as far as the following tests are concerned. The tests might be extended to ensure a specific version in a real-world implementation:

```
Describe 'PS developer workstation' {
    Context 'Packages' {
        It 'Chocolatey is installed' {
            { Get-Command choco -ErrorAction Stop } |
                Should -Not -Throw
        }

        It '<Name> is installed' -TestCases @(
            @{ Name = 'terraform' }
            @{ Name = 'git' }
        ) -Test {
            choco list -e $Name -l -r | Should -Match $Name
        }
    }
}
```

Each test case is defined as a Hashtable, and all keys in the Hashtable are available as variables inside the `It` statement automatically. This contrasts with Pester 4, which required a `param` block inside `It`.

The keys in the Hashtable can be used in the `It` description by enclosing the name in `< >`.

Pester allows the expansion of properties of values in the description. For instance, using `<Name.Length>` in the description would show the length of the string in the `Name` key. Not a very practical use in this case.

The outcome of running the tests can be viewed by saving the previous example to a file, such as `workstation.tests.ps1`, and using `Invoke-Pester`:

```
PS> Invoke-Pester -Path .\workstation.tests.ps1 -Output Detailed
Running tests from 'C:\workspace\workstation.tests.ps1'
Describing PS developer workstation
 Context Packages
   [+] Chocolatey is installed 20ms (15ms|5ms)
   [+] terraform is installed 802ms (799ms|3ms)
   [+] git is installed 786ms (786ms|1ms)
Tests completed in 1.79s
Tests Passed: 3, Failed: 0, Skipped: 0 NotRun: 0
```

The preceding tests are executed against a single subject, the local machine. You can use the `-ForEach` parameter of `Describe` or `Context` to run a set of tests against more than one subject.

Using the ForEach parameter

The `-ForEach` parameter can be used to execute either a `Describe` or `Context` block against an array of values. Continuing

with the theme of acceptance testing, it might be desirable to run a set of tests against several different servers.

The following tests assert that the DNS service exists on a set of Windows servers. The names of the servers are made up. The tests will potentially work if a meaningful set of names is provided:

```
Describe "DNS servers" -ForEach @(
    'dns01'
    'dns02'
) -Fixture {
    It "The DNS service is running on $_" {
        $params = @{
            ClassName    = 'Win32_Service'
            Filter       = 'Name="dns"'
            ComputerName = $_
        }
        Get-CimInstance @params | Should -Not -BeNullOrEmpty
    }
}
```

The `$_` variable used in the preceding example is created by Pester and is used to access each of the values in the `-ForEach` array in turn.

`Get-CimInstance` is used in the preceding example but `Invoke-Command` and `Get-Service` might be used instead if appropriate. You should implement tests in a way that is appropriate to the environment the test executes in.

All the tests used in this section have the potential to fail and raise errors that are not handled within the test. In the cases of the tests using `choco`, the tests will raise an error if Chocolatey is not actually installed. In the case of the last example, the tests will raise an error if the server does not exist or `Get-CimInstance` fails for any other reason.

Problems of this kind can potentially be handled by skipping tests or marking a test as inconclusive.

Conditional testing

There are two possible approaches for dealing with tests that cannot be executed:

- The result of `It` can be forcefully set by using `Set-ItResult`
- The test can be skipped entirely using the `-Skip` parameter

`Set-ItResult` can be used with the `<Name> is installed` test, enabling the test to account for situations where the `choco` command is not available.

Using Set-ItResult

The `Set-ItResult` command can be used inside any `It` statement. In the following example, it is used to change the result of the `It` statement based on the availability of the `choco` command:

```
Describe 'PS developer workstation' {
    Context 'Packages' {
        It '<Name> is installed' -TestCases @(
            @{ Name = 'terraform' }
            @{ Name = 'git' }
        ) -Test {
            if (Get-Command choco -ErrorAction SilentlyContinue) {
                choco list -e $Name -l -r | Should -Match $Name
            } else {
                Set-ItResult -Skipped
            }
        }
    }
}
```

The name of the test is changed in the result to show it has been skipped if the `choco` command is not available:

```
PS> Invoke-Pester -Path .\workstation.tests.ps1 -Output Detailed
Starting discovery in 1 files.
Discovering in C:\workspace\workstation.tests.ps1.
Found 2 tests. 27ms
Discovery finished in 31ms.
Running tests from 'C:\workspace\workstation.tests.ps1'
Describing PS developer workstation
 Context Packages
   [!] terraform is installed is skipped 14ms (9ms|5ms)
   [!] git is installed is skipped 2ms (1ms|1ms)
Tests completed in 217ms
Tests Passed: 0, Failed: 0, Skipped: 2 NotRun: 0
```

`Set-ItResult` allows the result to be set to `Inconclusive`, `Pending`, or `Skipped`.

The advantage of using `Set-ItResult`, in this case, is that the test cases are still processed. The `Name` value is expanded in the output of the tests. The `-Skip` parameter will stop Pester from expanding this value.

You can use the `-Skip` parameter on any `It` block that is not using `-TestCases`.

Using Skip

You can use `-Skip`, a switch parameter, to bypass one or more tests.

You can set an explicit value to the parameter such as a value based on a variable. The following example changes the test for the installation of Chocolatey. It will be skipped if the current operating system is not Windows:

```
Describe 'PS developer workstation' {
    Context 'Packages' {
        It 'Chocolatey is installed' -Skip:(-not $IsWindows) {
            { Get-Command choco -ErrorAction Stop } |
                Should -Not -Throw
        }
    }
}
```

The `$IsWindows`, `$IsMacOS`, and `$IsLinux` variables are all automatically available in PowerShell 6 and above.

Values used with the `-Skip` parameter must be available during the Discovery phase in Pester.

Pester phases

Pester 5 introduces the concept of different phases when executing tests. This is visible in the output of the tests run in this section:

```
Starting discovery in 1 files.
Discovering in C:\workspace\workstation.tests.ps1.
Found 2 tests. 27ms
Discovery finished in 31ms.
Running tests from 'C:\workspace\workstation.tests.ps1'
Describing PS developer workstation
```

First, the Discovery phase is run. During the Discovery phase, Pester attempts to find all the tests it will be running. Each test is defined by an `It` statement.

The Run phase will execute only those tests found during the Discovery phase.

This concept is new in Pester 5; it means that everything used to define what will be tested must be in place before discovery occurs, which affects the dynamic creation of tests.

Pester provides a `BeforeDiscovery` block, which may be placed either before or inside `Describe`. The code in `BeforeDiscovery` is, as the name suggests, executed before the Discovery run starts.

The last example used the predefined `$IsWindows` variable. As this is built-in, it is automatically available during the Discovery phase.

If the tests were instead to be executed based on a user-defined variable, this variable would need creating in `BeforeDiscovery`. The same limitation applies to any test cases used with the `-TestCases` parameter, and any arrays used with `-ForEach` parameters.

An earlier example used `-ForEach` to attempt to execute tests on an array of server names. If the server names had to be read from an external system, then that discovery action would be placed in the `BeforeDiscovery` block.

The following example uses the `Get-ADComputer` command from the `ActiveDirectory` module to get the list of servers to query. These tests will only succeed if the `ActiveDirectory` module is installed, and it is able to find server names to test:

```
BeforeDiscovery {
    $dnsServers = Get-ADComputer -Filter 'name -like "dns*"'
}
Describe "DNS servers" -ForEach $dnsServers -Fixture {
    It "The DNS service is running on $($_.Name)" {
        $params = @{
            ClassName    = 'Win32_Service'
            Filter       = 'Name="dns"'
            ComputerName = $_.DnsHostName
        }
        Get-CimInstance @params | Should -Not -BeNullOrEmpty
    }
}
```

`BeforeDiscovery` is therefore useful to ensure that the values needed to define which tests are present are in place when Pester is attempting to discover which tests are going to be executed.

Pester provides other named blocks to execute code at certain points during the Run phase. These can be used to define variables and set up conditions for tests. They should avoid defining which tests are executed.

Before and After blocks

Pester offers several blocks that you can use to perform actions before and after tests execute. Such blocks may be used to set up an environment or tear it down afterwards.

The blocks are:

- `BeforeAll`
- `BeforeEach`
- `AfterAll`
- `AfterEach`

The `All` blocks execute once, before (or after) any of the tests in that block. The `Each` blocks execute before (or after) individual `It` blocks.

Each block can exist once in any `Describe` or `Context` block. If a `Describe` block contains `BeforeAll`, and a nested `Context` also contains `BeforeAll` then both blocks will be executed (the `Describe` instance first, then the `Context` instance).

The `BeforeAll` and `BeforeEach` blocks are frequently used when defining a `Mock` for a command in unit testing.

Mocking commands

Mocking is used to reduce the scope of a set of tests and a vital part of unit testing. Mocking allows the implementation of a

command to be replaced with one defined in a test. Mocked commands are created using the `Mock` keyword.

The `Mock` keyword may be used in `BeforeAll`, `BeforeEach`, or `It`.

The following command reads a CSV file, then either starts or stops a service based on whether that service matches the state in the file.

```
@'
function Set-ServiceState {
    [CmdletBinding()]
    param (
        [string]$Path
    )
    Import-Csv $Path | ForEach-Object {
        $service = Get-Service $_.Name
        if ($service.Status -ne $_.ExpectedStatus) {
            if ($_.ExpectedStatus -eq 'Stopped') {
                Stop-Service -Name $_.Name
            } else {
                Start-Service -Name $_.Name
            }
        }
    }
}
'@ | Set-Content -Path module.psm1
```

Place this function in a file named `module.psm1`; it will be used as the subject of the first set of unit tests.

To effectively test this command, the system running the tests would need to have all the services listed in the `CSV` file, and it would have to be possible to change the state of those services.

Instead, depending on a complete system to execute on, the results of the code can be tested by mocking each of the commands external to the function.

Two of the commands (`Import-Csv` and `Get-Service)` must return information, and two (`Start-Service` and `Stop-Service`) return nothing at all.

Using `Mock` for `Start-Service` and `Stop-Service` is therefore straightforward:

```
Mock Start-Service
Mock Stop-Service
```

The output from `Import-Csv` and `Get-Service` needs to resemble the output from those real commands. The output can be simplified depending on what the command is expecting to do with that.

`Import-Csv` is expected to output an object with a `Name` and `ExpectedStatus` property:

```
Mock Import-Csv {
    [PSCustomObject]@{
        Name = 'service1'
        ExpectedStatus = 'Running'
    }
}
```

`Get-Service` is expected to return an object with a `Status` property, but no other properties from `Get-Service` are used. Mocking `Get-Service` allows the tests to run even if the current computer does not have the service being tested:

```
Mock Get-Service {
    [PSCustomObject]@{
        Status = 'Stopped'
    }
}
```

The name of the service is a parameter value for `Get-Service` and it is not used by the function. The name of the service can therefore be ignored in the object the mock emits.

Given the outputs that have been defined above, the expectation is that when running `Set-ServiceStatus`, you will use the `Start-Service` command to start `service1`.

Execution of the mock can be tested by using a `Should -Invoke` assertion:

```
@'
BeforeDiscovery {
    Import-Module .\module.psm1 -Force
}
Describe Set-ServiceState {
   BeforeAll {
        Mock Get-Service -MockWith {
            [PSCustomObject]@{
                Status = 'Stopped'
            }
        }
        Mock Import-Csv -MockWith {
            [PSCustomObject]@{
                Name           = 'service1'
                ExpectedStatus = 'Running'
            }
        }
        Mock Start-Service
        Mock Stop-Service
    }
    It 'When ExpectedStatus is running, starts the service' {
        Set-ServiceState -Path file.csv
        Should -Invoke Start-Service
    }
}
'@ | Set-Content Set-ServiceState.tests.ps1
```

The previous command saves the tests in a `Set-ServiceState.tests.ps1` file. The content of this file is modified during this section; the size of the file prohibits repeating the content in full for each change.

As `Import-Csv` is being mocked in the tests, the name used for the file (the `Path` parameter) is not relevant and you can use a made-up value.

The result of running the tests is shown here:

```
PS> Invoke-Pester -Path .\Set-ServiceState.tests.ps1
Starting discovery in 1 files.
Discovery finished in 10ms.
[+] C:\workspace\Set-ServiceState.tests.ps1 127ms (16ms|102ms)
Tests completed in 130ms
Tests Passed: 1, Failed: 0, Skipped: 0 NotRun: 0
```

In this function, there are three possible paths through the code for each service:

1. The service is in the expected state and neither `Start-Service` or `Stop-Service` run.
2. The service is stopped but was expected to be running, and `Start-Service` should run.
3. The service is running but was expected to be stopped, and `Stop-Service` should run.

A parameter filter for the `Get-Service` mock can be created to allow a different output based on the service name, which will allow each of the paths to be tested.

Parameter filtering

You can apply parameter filters to define when that mock should be used. Parameter filters are added using the `-ParameterFilter` parameter for `Mock`. The parameter filter is a script block that is most often used to test a parameter value used when calling the mock.

First, the mock for `Import-Csv` can be extended by adding two more services:

```
Mock Import-Csv -MockWith {
    [PSCustomObject]@{
        Name           = 'service1'
        ExpectedStatus = 'Running'
    }
    [PSCustomObject]@{
        Name           = 'service2'
        ExpectedStatus = 'Running'
    }
    [PSCustomObject]@{
        Name           = 'service3'
        ExpectedStatus = 'Stopped'
    }
}
```

Then the original mock for `Get-Service` is replaced with three new mocks. Each uses a `-ParameterFilter` and tests a different service name:

```
Mock Get-Service -ParameterFilter {
    $Name -eq 'service1'
} -MockWith {
    [PSCustomObject]@{
        Status = 'Running'
    }
}
Mock Get-Service -ParameterFilter {
    $Name -eq 'service2'
} -MockWith {
    [PSCustomObject]@{
        Status = 'Stopped'
    }
}
Mock Get-Service -ParameterFilter {
    $Name -eq 'service3'
} -MockWith {
    [PSCustomObject]@{
        Status = 'Running'
    }
}
```

Finally, the `It` block is adjusted. For this version, `Start-Service` will run once, and `Stop-Service` will run once. The previous assertion simply stated that `Start-Service` would run, which implicitly means it runs one or more times:

```
It 'Ensures all services are in the desired state' {
    Set-ServiceState -Path file.csv
    Should -Invoke Start-Service -Times 1
    Should -Invoke Start-Service -Times 1
}
```

Once the changes are made to the tests file, the single test will pass. However, while this test passes, it remains difficult to

explicitly relate cause to effect. A failure in any one or more of the comparisons will cause the preceding tests to fail, but it will not indicate which value caused the failure.

Instead of using `-ParameterFilter`, a more robust approach, in this case, might be to use `Context` to change the values provided by `Import-Csv` or the values returned by `Get-Service`.

Overriding mocks

You can use `Mock` in either an `It` or a `Context` block to override an existing mock or create new mocks which are specific to a specific branch of code. `Mock` is scoped to the block it is created in, therefore a `Mock` created in `It` only applies to that single `It` block.

Generally, the safest approach is to define default mocks under a `BeforeAll` in `Describe`, then to override those as needed. The presence of the default mocks acts as a safeguard. Running the subject of the tests, `Set-ServiceState`, in the following example, will only ever call a mocked command. It will never accidentally call the real command because something has been missed from a specific context:

```
@'
BeforeDiscovery {
    Import-Module .\module.psm1 -Force
}
Describe Set-ServiceState {
   BeforeAll {
        Mock Get-Service -MockWith {
            [PSCustomObject]@{
                Status = 'Running'
            }
        }
        Mock Import-Csv -MockWith {
            [PSCustomObject]@{
                Name           = 'service1'
                ExpectedStatus = 'Running'
            }
        }
        Mock Start-Service
        Mock Stop-Service
    }
}
'@ | Set-Content Set-ServiceState.tests.ps1
```

The first path through the code, when the service is already in the expected state and neither `Start-Service` or `Stop-Service` will be called, can be tested using the default mocks established above. The `It` block can explicitly assert that `Start-Service` and `Stop-Service` were not called:

```
It 'Service is running, expected running' {
    Set-ServiceState -Path file.csv
    Should -Invoke Start-Service -Times 0
    Should -Invoke Stop-Service -Times 0
}
```

The second path, when the service is stopped and should be started, can be achieved by overriding the mock for `Get-Service`. Note that the `Set-ServiceState` command is called again after overriding the mock:

```
It 'Service is stopped, expected running' {
    Mock Get-Service -MockWith {
        [PSCustomObject]@{
            Status = 'Stopped'
        }
    }
    Set-ServiceState -Path file.csv
    Should -Invoke Start-Service -Times 1
    Should -Invoke Stop-Service -Times 0
}
```

Finally, the last path runs when the service is running, but the expected state is stopped. This time the mock for `Import-Csv` is replaced:

```
It 'Service is running, expected stopped' {
    Mock Import-Csv -MockWith {
        [PSCustomObject]@{
            Name           = 'service1'
            ExpectedStatus = 'Stopped'
        }
    }
    Set-ServiceState -Path file.csv
    Should -Invoke Start-Service -Times 0
    Should -Invoke Stop-Service -Times 1
}
```

These new tests should be added to the `Describe` block of `Set-ServiceState.tests.ps1`. Once added, the tests file can be run with detailed output:

```
PS> $params = @{
>>     Path   = '.\Set-ServiceState.tests.ps1'
>>     Output = 'Detailed'
>> }
PS> Invoke-Pester @params
Starting discovery in 1 files.
Discovering in C:\workspace\Set-ServiceState.tests.ps1.
Found 3 tests. 11ms
Discovery finished in 16ms.
Running tests from 'C:\workspace\Set-ServiceState.tests.ps1'
Describing Set-ServiceState
  [+] Service is running, expected running 16ms (13ms|3ms)
  [+] Service is stopped, expected running 15ms (15ms|1ms)
  [+] Service is running, expected stopped 24ms (22ms|1ms)
Tests completed in 188ms
Tests Passed: 3, Failed: 0, Skipped: 0 NotRun: 0
```

These new tests provide a granular view of the different behaviors of the function. If a test fails, it is extremely easy to attribute cause to effect without having to spend extra time figuring out where either the tests or the subject failed.

The examples used to demonstrate mocking so far assume that the command being mocked is available on the current system. Commands that are not locally installed cannot be mocked.

Mocking non-local commands

If a command is not available on the system running tests, the attempt to create a mock will fail.

It is possible to work around this limitation with a small part of the command required by the tests. This can be referred to as a stub and typically consists of a function with only a parameter block.

The stub is used to provide something to mock, and the mock is used to track the execution of the function and ensure a subject behaves as intended.

For example, consider a function that creates and configures a DNS zone with a predefined set of parameter values:

```
function New-DnsZone {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory)]
        [String]$Name
    )
    $params = @{
        Name             = $Name
        DynamicUpdate    = 'Secure'
        ReplicationScope = 'Domain'
    }
    $zone = Get-DnsServerZone $Name -ErrorAction SilentlyContinue
    if (-not $zone) {
        Add-DnsServerPrimaryZone @params
    }
}
```

It may not be desirable to install the `DnsServer` tools on a development system to run unit tests. To mock and verify that `Add-DnsServerPrimaryZone` is called, a function must be created first:

```
Describe CreateDnsZone {
    BeforeAll {
        function Get-DnsServerZone { }
        function Add-DnsServerPrimaryZone { }
        Mock Get-DnsServerZone
        Mock Add-DnsServerPrimaryZone
    }
    It 'When the zone does not exist, creates a zone' {
        New-DnsZone -Name name
        Assert-MockCalled Add-DnsServerPrimaryZone
    }
}
```

Creating the function as shown here is enough to satisfy the tests, but the approach is basic. The test will pass even if parameter names are incorrect or missing.

A more advanced function to mock may be created by visiting a system with the command installed and retrieving the `param` block. The `ProxyCommand` type in PowerShell to get the `param` block from a system with the `DnsServer` module installed, for example:

```
using namespace System.Management.Automation
$command = Get-Command Add-DnsServerPrimaryZone
[ProxyCommand]::GetParamBlock(
    $command
)
```

For `Add-DnsServerPrimaryZone` the result is long. A command such as `Select-Object` has a simpler `param` block and is therefore easier to view. The first two parameters for `Select-Object` are shown here after running the `GetParamBlock` method:

```
PS> using namespace System.Management.Automation
PS> [ProxyCommand]::GetParamBlock((Get-Command Select-Object))
    [Parameter(ValueFromPipeline=$true)]
    [psobject]
    ${InputObject},
    [Parameter(ParameterSetName='DefaultParameter', Position=0)]
    [Parameter(ParameterSetName='SkipLastParameter', Position=0)]
    [System.Object[]]
    ${Property},
```

You can use this technique to create a stub of many modules, allowing tests to run even if the module is not locally installed.

The following snippet combines the `GetParamBlock` with `GetCmdletBindingAttribute` to create an accurate copy of the basics of the module to use as the basis for mocking a command.

```
using namespace System.Management.Automation
$moduleName = 'DnsServer'
Get-Command -Module $moduleName | ForEach-Object {
    $param = [ProxyCommand]::GetParamBlock($command)
    $param = $param -split '\r?\n' -replace '^\s{4}', '$0$0'
    'function {0} {{' -f $_.Name
    '    {0}' -f [ProxyCommand]::GetCmdletBindingAttribute($_)
    '    param ('
    $param
    '    )'
    '}'
    ''
} | Set-Content "$moduleName.psm1"
```

This approach works for the `DnsServer` module because the module is based on CIM classes; it only depends on assemblies that are already available in PowerShell.

Adding a copy of a module will improve the overall quality of the tests for a command. Tests will fail if a non-existent parameter is used, or if an invalid parameter combination is used.

Each of the mocks used so far has emitted a `PSCustomObject`, and in many cases a `PSCustomObject` is enough to use within a set of tests.

Mocking objects

Mocking allows the result of running another command to be faked. The examples in the previous section have returned a `PSCustomObject` where output is required.

It is not uncommon for a command to expect to work with the properties and methods of another object. This might be a value returned by another command, or it might be the value of a parameter the test subject requires.

The ability to mock objects or a specific type or objects that implement methods is important in testing.

Two approaches can be taken when testing:

- Methods can be added to a `PSCustomObject`
- .NET types can be disarmed and returned

PowerShell includes many modules that are based on CIM classes. These modules often expect CIM instances as input to work. Testing code that uses CIM-based commands may need to create values that closely resemble the real command output.

Methods can be added to a `PSCustomObject`, allowing code that uses those methods to be tested without needing to use a more specific .NET type.

Adding methods to PSCustomObject

Objects with specific properties can be simulated by creating a `PSCustomObject` object:

```
[PSCustomObject]@{
    Property = "Value"
}
```

If the subject of a test takes the result of a mocked command and invokes a method, it will fail unless the method is available. You can add methods to a `PSCustomObject` using `Add-Member`:

```
$object = [PSCustomObject]@{} |
    Add-Member MethodName -MemberType ScriptMethod -Value { }
$object
```

If the method used already exists, such as the `ToString` method, then the `-Force` parameter must be used:

```
$object = [PSCustomObject]@{}
$object |
    Add-Member ToString -MemberType ScriptMethod -Force -Value { }
$object
```

As many methods as needed can be added to the `PSCustomObject` as required.

The method added to the `PSCustomObject` may return nothing (as in the preceding examples), return a specific value, or set a variable in script scope which can be tracked or tested. This idea is explored when disarming an existing .NET object.

Disarming .NET types

A piece of code being tested may interact with a specific .NET type. The .NET type may (by default) need to interact with other systems in a way that is not desirable when testing.

The following simple function expects to receive an instance of a `SqlConnection` object and expects to be able to call the `Open` method:

```
using namespace System.Data.SqlClient
function Open-SqlConnection {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory)]
        [SqlConnection]$SqlConnection
    )
    if ($sqlConnection.State -eq 'Closed') {
        $SqlConnection.Open()
    }
}
```

As the value of the `SqlConnection` parameter is explicitly set to accept an instance of `System.Data.SqlClient.SqlConnection`, you cannot use a `PSCustomObject` as a substitute.

When running the function in a test, an instance of `SqlConnection` must be created to pass to the function.

The following `It` block creates such an instance and passes it to the function:

```
It 'Opens an SQL connection' {
    $connection = [System.Data.SqlClient.SqlConnection]::new()
    Open-SqlConnection -SqlConnection $connection
}
```

This `It` block does not contain any assertions yet. It can assert that no errors should be thrown, but the test can only succeed if the computer running the tests is running an SQL server instance. By extension, the test can only fail if the computer running the tests is not a SQL server.

Each of the following tests can be added to a `sql.tests.ps1` file. The `Describe` block has been omitted from the examples to reduce indentation. The following command creates the tests file. All the content of the `Describe` block should be replaced with each example.

```
@'
BeforeDiscovery {
function Script:Open-SqlConnection {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory)]
        [System.Data.SqlClient.SqlConnection]$SqlConnection
    )
    if ($sqlConnection.State -eq 'Closed') {
        $SqlConnection.Open()
    }
}
}
Describe Open-SqlConnection {
}
'@ | Set-Content sql.tests.ps1
```

PowerShell code will prefer to call a `ScriptMethod` on an object over a `Method` provided by the .NET type. Therefore, a you can create a disarmed version of the `SqlConnection` object using `Add-Member`:

```
BeforeAll {
    $connection = [System.Data.SqlClient.SqlConnection]::new()
    $connection |
        Add-Member Open -MemberType ScriptMethod -Value { } -Force
}
It 'Opens an SQL connection' {
    Open-SqlConnection -SqlConnection $connection
}
```

This step solves the problem of needing a SQL server to run `Open`, but it does not solve the problem of testing if `Open` was called. After all, the command does not return the connection object; there is no output to test.

You can solve the problem by making the `Open` method do something. Possible actions include:

- Return a value
- Set a scoped variable
- Set a property on the `SqlConnection` object

The `Open` method, by default, does not return a value, and the statement that calls `Open` in the function is not assigned. A value could simply be returned by the method and tested:

```
BeforeAll {
    $connection = [System.Data.SqlClient.SqlConnection]::new()
    $connection |
        Add-Member Open -MemberType ScriptMethod -Force -Value {
            $true
        }
}
It 'Opens an SQL connection' {    Open-SqlConnection -SqlConnection $connection |
        Should -BeTrue
}
```

In some cases, such an approach might fail or become overly complex. For instance, if this is only one small step the command takes and there are other outputs to consider.

Setting a variable in script scope in the mocked method will allow a test to see if the method is executed. In this case, the scoped value might be reset in a `BeforeEach` block, ensuring it is accurately recorded in each test:

```
BeforeAll {
    $connection = [System.Data.SqlClient.SqlConnection]::new()
    $connection |
        Add-Member Open -MemberType ScriptMethod -Force -Value {
            $Script:Opened = $true
        }
}
BeforeEach {
    $Script:Opened = $false
}
It 'Opens an SQL connection' {
    Open-SqlConnection -SqlConnection $connection
    $Script:Opened | Should -BeTrue
}
```

Finally, a property of the `SqlConnection` object might be set. The real `Open` method sets the value of the `State` property. This property is read-only and cannot be set directly. A `NoteProperty` must be added for `Open` to change.

```
BeforeAll {
    $connection = [System.Data.SqlClient.SqlConnection]::new()
    $connection |
        Add-Member Open -MemberType ScriptMethod -Force -Value {
            $this.State = 'Open'
        }
    $connection |
        Add-Member State -NotePropertyValue Closed -Force
}
It 'Opens an SQL connection' {
    Open-SqlConnection -SqlConnection $connection
    $connection.State | Should -Be 'Open'
}
```

Any of the preceding options might be used when mocking methods and properties on .NET objects.

The approach can be taken further still by using the `New-MockObject` command in Pester. `New-MockObject` creates an instance of a .NET type with no code behind it at all.

`New-MockObject` is not appropriate in all cases. If you use this command to create the `SqlConnection` object used above, attempting to call the real `Open` method will always raise an error:

```
PS> $connection = New-MockObject System.Data.SqlClient.SqlConnection
PS> $connection.Open()
MethodInvocationException: Exception calling "Open" with "0" argument(s): "Object reference not set to
an instance of an object."
```

You can apply the techniques used in the previous examples to a wide variety of .NET objects. `CimInstance` objects are a special case when it comes to mocking.

Mocking CIM objects

Many modules in PowerShell are based on CIM classes. For example, the `Net` modules, such as `NetAdapter`, `NetSecurity`, and `NetTCPIP`, are all based on CIM classes.

The commands in these modules either return CIM instances or include parameters that require a specific CIM instance as an argument.

For example, the following function uses two of the commands in a pipeline. Any tests would have to account for the CIM classes when mocking commands:

```
function Enable-PhysicalAdapter {
    Get-NetAdapter -Physical | Enable-NetAdapter
}
```

When these commands act in a pipeline, `Enable-NetAdapter` fills the `InputObject` parameter from the pipeline. `Get-Help` shows that the parameter accepts an array of `CimInstance` from the pipeline:

```
PS> Get-Help Enable-NetAdapter -Parameter InputObject
-InputObject <CimInstance[]>
    Specifies the input to this cmdlet. You can use this parameter, or you can pipe the input to this
cmdlet.
    Required?                    true
    Position?                    named
    Default value
    Accept pipeline input?       true (ByValue)
    Accept wildcard characters?  false
```

However, this is not the whole story. The parameter value is further constrained by a `PSTypeName` attribute. This can be seen using `Get-Command`:

```
$command = (Get-Command Enable-NetAdapter)
$parameter = $command.Parameters['InputObject']
$attribute = $parameter.Attributes |
    Where-Object TypeId -match 'PSTypeName'
$attribute.PSTypeName
```

The result is the `PSTypeName` the command expects to receive from the pipeline:

```
Microsoft.Management.Infrastructure.CimInstance#MSFT_NetAdapter
```

Any mock for `Get-NetAdapter` must therefore return an `MSFT_NetAdapter` `CimInstance` object. Before the instance can be created, one final piece of information is required: the namespace of the CIM class.

The namespace can be taken from any object returned by `Get-NetAdapter`:

```
PS> Get-NetAdapter | Select-Object CimClass -First 1
CimClass
--------
ROOT/StandardCimv2:MSFT_NetAdapter
```

Finally, the `CimInstance` object can be created using the `New-CimInstance` command as shown here:

```
$params = @{
    ClassName  = 'MSFT_NetAdapter'
    Namespace  = 'ROOT/StandardCimv2'
    ClientOnly = $true
}
New-CimInstance @params
```

This instance can be added to a mock for `Get-NetAdapter` when testing the `Enable-PhysicalAdapter` command:

```
BeforeDiscovery {
    function Script:Enable-PhysicalAdapter {
        Get-NetAdapter -Physical | Enable-NetAdapter
    }
}
Describe Enable-PhysicalAdapter {
    BeforeAll {
        Mock Enable-NetAdapter
        Mock Get-NetAdapter {
            $params = @{
                 ClassName  = 'MSFT_NetAdapter'
                 Namespace  = 'ROOT/StandardCimv2'
                 ClientOnly = $true
            }
            New-CimInstance @params
        }
    }
    It 'Enables a physical network adapter' {
        { Enable-PhysicalAdapter } | Should -Not -Throw
        Should -Invoke Enable-NetAdapter -Times 1
    }
}
```

The commands used in each of the tests in this section are expected to be available in the global scope so that Pester can mock and run the commands. Pester is also able to test commands and classes that are not exported from a module.

InModuleScope

The `InModuleScope` command and the `-ModuleName` parameter of `Should` and `Mock` are important features of Pester. The command and parameters allow access to content that is normally in the module scope and inaccessible outside.

The following two commands were first introduced in *Chapter 20, Building Modules*:

```
@'
function GetRegistryParameter {
    [CmdletBinding()]
    param ( )
    @{
        Path = 'HKLM:\SYSTEM\CurrentControlSet\Services\LanmanServer\Parameters'
        Name = 'srvcomment'
    }
}
function Get-ComputerDescription {
    [CmdletBinding()]
    param ( )
    $getParams = GetRegistryParameter
    Get-ItemPropertyValue @getParams
}
Export-ModuleMember Get-ComputerDescription
'@ | Set-Content LocalMachine.psm1
```

The function `GetRegistryParameter` can be tested in Pester by using `InModuleScope`:

```
@'
BeforeDiscovery {
    Import-Module .\LocalMachine.psm1 -Force
}
Describe GetRegistryParameter {
    It 'Returns a hashtable' {
        InModuleScope -ModuleName LocalMachine {
            GetRegistryParameter
        } | Should -BeOfType [Hashtable]
    }
}
'@ | Set-Content GetRegistryParameter.tests.ps1
```

If the `InModuleScope` command is omitted, the test will fail and the `GetRegistryParameter` function is not exported from the module and is therefore not normally accessible.

The result of running the tests is shown here:

```
PS> Invoke-Pester -Script .\GetRegistryParameter.tests.ps1
Starting discovery in 1 files.
Discovery finished in 227ms.
Running tests.
[+] C:\workspace\GetRegistryParameter.tests.ps1 1.04s (152ms|702ms)
Tests completed in 1.06s
Tests Passed: 1, Failed: 0, Skipped: 0 NotRun: 0
```

A test document may include more than one use of `InModuleScope`, but it is advisable to keep the size of these blocks as small as possible. `InModuleScope` should not be used to enclose `Describe`, `Context`, and `It`.

In the same way, if it were desirable to mock that command when testing the `Get-ComputerDescription` command, the `-ModuleName` parameter is required for the `Mock` keyword:

```
BeforeAll {
    Mock GetRegistryParameter -ModuleName LocalMachine
}
```

`InModuleScope` can be used to access anything in the module scope, including private commands, classes, and enumerations, and any module-scoped variables.

Pester in scripts

Using `InModuleScope` can add complexity when running `Invoke-Pester` from a script.

When `Invoke-Pester` is run from a global scope, `-ModuleName` is only required to access private components of a module.

When `Invoke-Pester` is run from a script, problems may surface because the script scope breaks Pester's scoping.

Consider the following tests:

```
@'
BeforeDiscovery {
    Import-Module .\LocalMachine.psm1 -Force
}
Describe Get-ComputerDescription {
    BeforeAll {
        Mock Get-ItemPropertyValue {
            'Mocked description'
        }
    }
    It 'Returns the mocked description' {
        Get-ComputerDescription |
            Should -Be 'Mocked description'
        Should -Invoke Get-ItemPropertyValue
    }
}
'@ | Set-Content Get-ComputerDescription.tests.ps1
```

When `Invoke-Pester` is run from the console, the tests pass provided the `LocalMachine` module was successfully imported:

```
PS> Invoke-Pester -Path .\Get-ComputerDescription.tests.ps1
Starting discovery in 1 files.
Discovery finished in 7ms.
[+] C:\workspace\Get-ComputerDescription.tests.ps1 98ms (6ms|86ms)
Tests completed in 99ms
Tests Passed: 1, Failed: 0, Skipped: 0 NotRun: 0
```

If instead the `Invoke-Pester` command is put in a script, and the script is run the mock is completely ignored:

```
@'
Invoke-Pester -Path .\Get-ComputerDescription.tests.ps1
'@ | Set-Content script.ps1
```

This is shown here when running the script:

```
PS> .\script.ps1
Starting discovery in 1 files.
Discovery finished in 7ms.
[-] Get-ComputerDescription.Returns the mocked description 5ms (4ms|1ms)
 PSArgumentException: Property srvcomment does not exist at path
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\LanmanServer\Parameters.
 at Get-ComputerDescription, C:\workspace\LocalMachine.psm1:16
 at <ScriptBlock>, C:\workspace\Get-ComputerDescription.tests.ps1:12
Tests completed in 117ms
Tests Passed: 0, Failed: 1, Skipped: 0 NotRun: 0
```

To work around this problem, the `-ModuleName` parameter must be added to all `Mock` commands and all `Should -Invoke` assertions. In the following example, a splat is used:

```
Describe Get-ComputerDescription {
    BeforeAll {
        $module = @{
            ModuleName = 'LocalMachine'
        }
        Mock Get-ItemPropertyValue @module {
            'Mocked description'
        }
    }
    It 'Returns the mocked description' {
        Get-ComputerDescription |
            Should -Be 'Mocked description'
        Should -Invoke Get-ItemPropertyValue @module
    }
}
```

In all cases the module name is the scope the mock should be created in, so the subject module, not the module the mocked command belongs to.

With this in place, `Invoke-Pester` can be run from a script:

```
PS> .\script.ps1
Starting discovery in 1 files.
Discovery finished in 23ms.
[+] C:\workspace\Get-ComputerDescription.tests.ps1 135ms (25ms|88ms)
Tests completed in 137ms
Tests Passed: 1, Failed: 0, Skipped: 0 NotRun: 0
```

Pester is a wonderful tool for writing tests for a variety of different purposes. The tools above offer an introduction to the capabilities of the module.

Summary

This chapter explored the complex topic of testing in PowerShell.

Static analysis is one part of testing and is the approach used by modules like `PSScriptAnalyzer`. Static analysis makes use of the Abstract Syntax Tree and tokenizers in PowerShell.

The Abstract Syntax Tree or AST describes the content of a block of code as a tree of different elements, starting with a `ScriptBlockAst` at the highest level. You can use the `ParseInput` and `ParseFile` methods of the `Parser` type to get either an instance of the AST for a piece of code or the tokens that make up a script that includes comments.

The `ShowPSAst` module can be used to visualize and explore the AST tree. `ShowPSAst` is a useful tool when starting to work with AST as the tree can quickly become complex.

`PSScriptAnalyzer` uses either AST or tokens to define rules. Rules can be used to test and enforce personal or organization-specific practices.

Pester is a testing framework and this chapter explored both acceptance and unit testing.

Acceptance testing is commonly used to assess the state of systems and services. You can use Pester to define tests, which can be saved and shared. Such tests can be used to validate a system is configured or behaving as it should be.

Pester is a rich tool that supports iteration with the `-ForEach` or `-TestCases` parameters. Conditional testing can be achieved using the `Set-ItResult` and `-Skip` parameters.

Mocking is an exceptionally useful feature of Pester and is often used when writing unit tests to reduce the amount of code that must be tested when the subject is a single command.

The next chapter explores error handling in PowerShell, including terminating and non-terminating errors and the use of `try`, `catch`, `finally`, and the `trap` statement.

22

Error Handling

Errors communicate unexpected conditions and exceptional circumstances. Errors often contain useful information that you can use to diagnose a condition.

PowerShell has two different types of errors, terminating and non-terminating, and several different ways to raise and handle them.

Error handling in PowerShell is a complex topic, which is not helped by incorrect assertions in help documentation surrounding terminating errors. These challenges are explored in this chapter.

Self-contained blocks of code are described as scripts in this chapter. That is, functions, script blocks, and scripts can be considered interchangeable in the context of error handling.

This chapter covers the following topics:

- Error types
- Error actions
- Raising errors
- Catching errors

Error types

As mentioned above, PowerShell defines two different types of errors: terminating and non-terminating errors.

Each command in PowerShell may choose to raise either of these, depending on the operation.

Non-terminating errors

A non-terminating error, a type of informational output, is written without stopping a script. Non-terminating errors exist to allow a command to continue in the event of a partial failure. For example, if a command is acting on a pipeline and one item fails, the command can write an error and continue to process the remaining items.

Non-terminating errors can be written using the `Write-Error` command, although a deeper analysis of this approach and alternatives are explored later in this chapter.

The following example demonstrates how a non-terminating error can be used in a function accepting pipeline input:

```
function Update-Value {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory, ValueFromPipeline)]
        [string]$Value
    )
    process {
        if ($Value.Length -lt 5) {
            Write-Error ('The value {0} is unacceptable' -f $Value)
        } else {
            'Updated value: {0}' -f $Value
        }
    }
}
```

When executed on a pipeline containing words, the command will write an error and continue every time it encounters a word with a length less than `5`:

```
PS> 'value', 'val', 'longvalue' | Update-Value
Updated value: value
Update-Value: The value val is unacceptable
Updated value: longvalue
```

Scenarios like this are why non-terminating errors exist. It is then up to the consumer of the command to decide if a script should continue.

Non-terminating errors should be used when a script acts on more than one input, and an error for a single item does not affect success or failure for subsequent items.

Terminating errors are used to stop a script in the event of a problem.

Terminating errors

Terminating errors are used to stop an operation from continuing.

Terminating errors split in two depending on how the error is raised; this leads to some inconsistent behavior, which is explored in the *Raising errors* section later in this chapter.

A terminating error stops a script once an error is thrown. No further commands will execute after the error is thrown.

When running the following command, the second `Write-Host` statement will never execute:

```
function Stop-Command {
    Write-Host 'First'
    throw 'Error'
    Write-Host 'Second'
}
```

Running the function will show that the second `Write-Host` command does not execute:

```
PS> $ErrorActionPreference = 'Continue'
PS> Stop-Command
First
Exception:
Line |
   3 |      throw 'Error'
     |      ~~~~~~~~~~~~~
     | Error
```

The assertion that the second statement will never run has a caveat, terminating errors raised by `throw` are affected by the `$ErrorActionPreference` variable. This is the reason for ensuring `$ErrorActionPreference` is set to `Continue` before running the last example.

Setting the `ErrorAction` to `SilentlyContinue` shows that the error is ignored:

```
PS> $ErrorActionPreference = 'SilentlyContinue'
PS> Stop-Command
First
Second
```

This contradicts the statements in the PowerShell documentation:

- `Get-Help about_Throw` asserts that the `throw` keyword causes a terminating error
- `Get-Help about_Preference_Variables` asserts that `$ErrorActionPreference` and `-ErrorAction` do not affect terminating errors

Exactly how this affects code and how to create code that behaves consistently are explored in this chapter.

Before exploring raising errors in PowerShell, the `ErrorActionPreference` variable and `-ErrorAction` parameters should be introduced.

Error actions

The `-ErrorAction` parameter and the `ErrorActionPreference` variable are used to control what happens when a non-terminating error is encountered, subject to the previous notes about `throw`.

The `-ErrorAction` parameter is made available on a command when the `CmdletBinding` attribute is present. The `CmdletBinding` attribute is implicitly added if one or more parameters in a script use the `Parameter` attribute.

By default, `-ErrorAction` is set to `Continue`. Non-terminating errors will be displayed, but a script will continue to run.

`$ErrorActionPreference` is a scoped variable, which you can use to affect all commands in a particular scope and any child scopes. By default, `$ErrorActionPreference` is set to `Continue`. You can override the variable in child scopes (such as a function inside a script).

All errors in a session are implicitly added to the reserved variable `$Error` unless the error action is set to `Ignore`. Now, `$Error` is an ArrayList and contains each error in the session with the newest first (at index 0).

If `-ErrorAction` is set to `SilentlyContinue`, errors will still be added to the `$Error` automatic variable, but the error will not be displayed in the host.

The following function writes a non-terminating error using `Write-Error`:

```
function Start-Task {
    [CmdletBinding()]
    param ( )

    Write-Error 'Something went wrong'
}
Start-Task -ErrorAction SilentlyContinue
```

The error is not displayed in the host or console, the error may be viewed as the latest entry in the `$Error` variable:

```
PS> $Error[0]
Start-Task: Something went wrong
```

If the error action is set to `Stop`, a non-terminating error becomes a terminating error. In the console, the output in either case is similar. However, a terminating error is displayed differently when viewing `$Error[0]`:

```
PS> Start-Task -ErrorAction Stop
Start-Task: Something went wrong
PS> $Error[0]
ErrorRecord                 : Something went wrong
WasThrownFromThrowStatement : False
TargetSite                  : System.Collections.ObjectModel.Coll
                              ection`1[System.Management.Automati
                              on.PSObject] Invoke(System.Collecti
                              ons.IEnumerable)
StackTrace                  :    at System.Management.Automation.
                              Runspaces.PipelineBase.Invoke(IEnum
                              erable input)
                                 at System.Management.Automation.
                              Runspaces.Pipeline.Invoke()
                                 at Microsoft.PowerShell.Executor
                              .ExecuteCommandHelper(Pipeline
                              tempPipeline, Exception&
                              exceptionThrown, ExecutionOptions
                              options)
Message                     : The running command stopped
                              because the preference variable
                              "ErrorActionPreference" or common
                              parameter is set to Stop:
                              Something went wrong
Data                        : {System.Management.Automation.Inter
                              preter.InterpretedFrameInfo}
InnerException              :
HelpLink                    :
Source                      : System.Management.Automation
HResult                     : -2146233087
```

The content of the object is difficult to read, especially in a narrow console.

PowerShell 7 includes a `Get-Error` command, which you can use to explore errors that have been raised in greater detail.

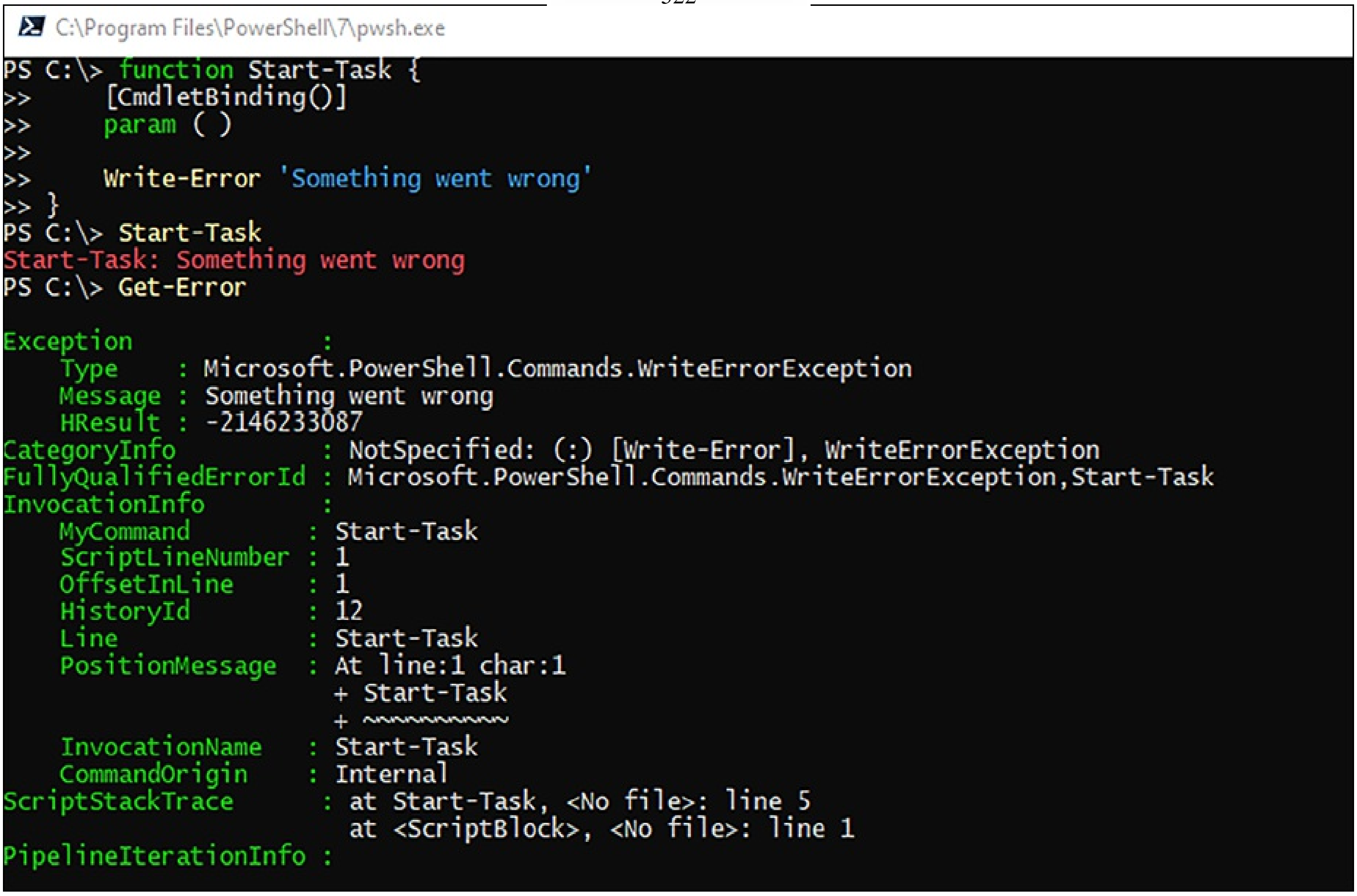
About Get-Error

The `Get-Error` command was introduced with PowerShell 7. When used without any parameters, it gets the latest error from the `$Error` variable and provides a summary of the content of that error.

If the `Start-Task` function above is executed, `Get-Error` can be used to explore the details of the error it raised.

The output differs between terminating and non-terminating errors, more information is shown for terminating errors.

Figure 22.1 shows the output of the command when a non-terminating error has been written; the content does not display well in a narrow console:



```
C:\Program Files\PowerShell\7\pwsh.exe
PS C:\> function Start-Task {
>>     [CmdletBinding()]
>>     param ( )
>>
>>     Write-Error 'Something went wrong'
>> }
PS C:\> Start-Task
Start-Task: Something went wrong
PS C:\> Get-Error

Exception             :
    Type    : Microsoft.PowerShell.Commands.WriteErrorException
    Message : Something went wrong
    HResult : -2146233087
CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorException
FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException,Start-Task
InvocationInfo        :
    MyCommand        : Start-Task
    ScriptLineNumber : 1
    OffsetInLine     : 1
    HistoryId        : 12
    Line             : Start-Task
    PositionMessage  : At line:1 char:1
                       + Start-Task
                       + ~~~~~~~~~~
    InvocationName   : Start-Task
    CommandOrigin    : Internal
ScriptStackTrace      : at Start-Task, <No file>: line 5
                        at <ScriptBlock>, <No file>: line 1
PipelineIterationInfo :
```

Figure 22.1: Get-Error command

You can use `Get-Error` to view the last error, a user-defined number of the newest errors, and a specific error using `ErrorRecord`.

For example, if `$Error` contained 10 errors, and only one of those from the middle of the set was of interest, `Get-Error` would accept one or more errors as pipeline input:

```
$Error[4] | Get-Error
```

If there were fewer than 4 errors, another error would be written.

When writing scripts, it is often desirable to write or raise an error.

Raising errors

When writing a script, it may be desirable to use errors to notify the person running the script of a problem. The severity of the problem will dictate whether an error is non-terminating or terminating.

If a script makes a single change to many diverse, unrelated objects, a terminating error might be frustrating for anyone using the script.

On the other hand, if a script fails to read a critical configuration file, a terminating error is likely the right choice.

Error records

When an error is raised in PowerShell, an `ErrorRecord` object is created (explicitly or implicitly).

An `ErrorRecord` object contains several fields that are useful for diagnosing an error. `ErrorRecord` can be explored using `Get-Member` or by using the `Get-Error` command.

For example, an `ErrorRecord` will be generated when attempting to divide by 0:

```
100 / 0
$record = $Error[0]
```

The `ErrorRecord` object that was generated includes `ScriptStackTrace`, which includes the script names that were called and the position where an error occurred. `ScriptTrackTrace` is extremely useful when debugging problems in larger scripts:

```
PS> $record.ScriptStackTrace
at <ScriptBlock>, <No file>: line 1
```

A bespoke `ErrorRecord` can be created within a script to more clearly describe a problem to the user. The error record might include additional information to assist with debugging.

For example, if the values for a division operation were dynamically set, an `ErrorRecord` might be created to include those values in the `TargetObject` to assist with debugging:

```
using namespace System.Management.Automation
$numerator = 10
$denominator = 0
try {
    $numerator / $denominator
} catch {
    $errorRecord = [ErrorRecord]::new(
        [Exception]::new($_.Exception.Message),
        'InvalidDivision',    # ErrorId
        'InvalidOperation',   # ErrorCategory
        [PSCustomObject]@{  # TargetObject
            Numerator   = $numerator
            Denominator = $denominator
        }
    )
    Write-Error -ErrorRecord $errorRecord
}
```

The `ErrorRecord` object and the constructor in the preceding example are described on Microsoft Docs: https://docs.microsoft.com/dotnet/api/system.management.automation.errorrecord.-ctor?view=powershellsdk-7.0.0#System_Management_Automation_ErrorRecord__ctor_System_Exception_System_String_System_Management_Automation_ErrorCategory_System_Object_.

The values added to the `ErrorRecord` may be viewed by exploring the `TargetObject` property:

```
PS> $Error[0].TargetObject
Numerator     Denominator
---------     -----------
       10               0
```

The `try-catch` statement used in the previous example is covered in detail while exploring `try`, `catch`, and `finally` later in this chapter.

Raising non-terminating errors

You can use the `Write-Error` command to write a non-terminating error message.

The `Write-Error` command can display a simple message:

```
Write-Error -Message 'Message'
```

Or the error might include additional information, such as a category and error ID to aid diagnosis by the person using the

script:

```
$params = @{
    Message  = 'Message'
    Category = 'InvalidOperation'
    ErrorID  = 'UniqueID'
}
Write-Error @params
```

The following example shows a non-terminating error that was raised while running a loop:

```
function Test-Error {
    [CmdletBinding()]
    param ( )
    for ($i = 0; $i -lt 3; $i++) {
        Write-Error -Message "Iteration: $i"
    }
}
```

When the function runs, an error is displayed three times without stopping execution:

```
PS> Test-Error
Test-Error: Iteration: 0
Test-Error: Iteration: 1
Test-Error: Iteration: 2
```

Setting the value of `ErrorAction` to `Stop` will cause `Write-Error` to throw a terminating error, ending the function within the first iteration of the loop:

```
PS> Test-Error -ErrorAction Stop
Test-Error: Iteration: 0
```

Alternatively, the error can be silenced (`SilentlyContinue`) or ignored (`Ignore`), depending on the importance of the error to the user of the function.

One disadvantage of using the `Write-Error` command is that it does not set the value of the automatic variable `$?` unless `ErrorAction` is set to `Stop` . The variable `$?` is set to `True` if the last command was successful, or `False` if the last command failed:

```
PS> Test-Error
Test-Error: Iteration: 0
Test-Error: Iteration: 1
Test-Error: Iteration: 2
PS> $?
True
```

The `WriteError` method may be used as an alternative to the `Write-Error` command.

Using the WriteError method

The `WriteError` method is used by binary commands and can optionally be used by scripts, provided the script includes the `CmdletBinding` attribute.

The presence of the `CmdletBinding` attribute makes the `$PSCmdlet` variable available for use within a script. The `$PSCmdlet` variable provides access to the `WriteError` method.

The `WriteError` method requires an `ErrorRecord` as an argument as shown in the following example:

```
using namespace System.Management.Automation
function Test-Error {
    [CmdletBinding()]
    param ( )
    for ($i = 0; $i -lt 3; $i++) {
        $errorRecord = [ErrorRecord]::new(
            [Exception]::new('Iteration {0}' -f $i),
            'InvalidOperation',
            'InvalidOperation',
            $i
        )
        $PSCmdlet.WriteError($errorRecord)
    }
}
```

When running the command, the value of `$?` correctly reflects that one or more errors occurred in the last command:

```
PS> Test-Error
Test-Error: Iteration 0
Test-Error: Iteration 1
Test-Error: Iteration 2
PS> $?
False
```

You can raise terminating errors by setting `-ErrorAction` to `Stop` when using the `Write-Error` command. However, it is often more appropriate to explicitly raise a terminating error.

Raising terminating errors

The `throw` keyword raises a terminating error, as in the following example:

```
throw 'Error message'
```

Existing exception types are documented in .NET Framework; each is ultimately derived from the `System.Exception` type found in the .NET reference: https://docs.microsoft.com/dotnet/api/system.exception.

`throw` may be used with a string or a message, as shown previously. `throw` may also be used with an exception object:

```
throw [ArgumentException]::new('Unsupported value')
```

Or you can use `throw` with `ErrorRecord`:

```
using namespace System.Management.Automation
throw [ErrorRecord]::new(
    [InvalidOperationException]::new('Invalid operation'),
    'AnErrorID',
    [ErrorCategory]::InvalidOperation,
    $null
)
```

Commands in binary modules cannot use `throw`; it has a different meaning in the languages that might be used to author a cmdlet (such as C#, VB, or F#). Cmdlets use the `ThrowTerminatingError` method instead.

Using the ThrowTerminatingError method

Like the `WriteError` method, `ThrowTerminatingError` is made available to scripts that use the `CmdletBinding` attribute. The `ThrowTerminatingError` method is available on the `$PSCmdlet` variable.

The `ThrowTerminatingError` method requires an `ErrorRecord` object as shown in the following example:

```
using namespace System.Management.Automation
function Invoke-Something {
    [CmdletBinding()]
    param ( )

    $errorRecord = [ErrorRecord]::new(
        [InvalidOperationException]::new('Failed'),
        'AnErrorID',
        [ErrorCategory]::OperationStopped,
        $null
    )
    $PSCmdlet.ThrowTerminatingError($errorRecord)
}
```

Running the command will show this simple error message:

```
PS> Invoke-Something
Invoke-Something: Failed
```

More detailed error information can be viewed using the `Get-Error` command.

An error may only exist to communicate a problem to an end user, but it is common to need to capture and handle errors in another script.

Catching errors

Capturing an error so that a script can react to that error depends on the error type.

- Non-terminating errors can be captured by using `ErrorVariable`
- Terminating errors can be captured using either a `try`, `catch`, and `finally` statement, or by using a `trap` statement

The `-ErrorVariable` parameter can be used to create a scoped alternative to the `$Error` variable.

ErrorVariable

The `$Error` variable is a collection (ArrayList) of handled and unhandled errors raised in the PowerShell session.

You can use the `-ErrorVariable` parameter to name a variable that should be used for a specific script. The `ErrorVariable` accepts the name of a variable and is created as an ArrayList.

The following function writes a single error using the `Write-Error` command:

```
function Invoke-Something {
    [CmdletBinding()]
    param ( )

    Write-Error 'Invoke-Something Failed'
}
```

The command can be run using the `-ErrorVariable` parameter. The `-ErrorAction` parameter below is used to suppress the normal error output:

```
$params = @{
    ErrorVariable = 'MyErrorVariable'
    ErrorAction   = 'SilentlyContinue'
}
Invoke-Something @params
```

Nothing is displayed in the console, but the error is added to `$MyErrorVariable`:

```
PS> $MyErrorVariable
Invoke-Something: Invoke-Something Failed
```

Error messages written to an `-ErrorVariable` are duplicated in `$Error`.

An -ErrorVariable is never null

If no errors occur, a variable will still be created as an ArrayList, but the list will contain no elements.

The `Count` property might be inspected to see if any errors occurred:

```
$MyErrorVariable.Count -eq 0
```

A single `-ErrorVariable` can be shared by several different scripts; by default the content of the variable is overwritten. If `+` is added to the beginning of the variable name, then new errors will be added without overwriting the content of the variable. The following command is run twice: the first run creates a new error variable, and the second adds to that existing variable:

```
Invoke-Something -ErrorVariable MyErrorVariable
Invoke-Something -ErrorVariable +MyErrorVariable
```

Once the two commands have finished, `$MyErrorVariable` will contain two errors, one from each command.

Terminating errors are most frequently handled using `try`, `catch`, and `finally`.

try, catch, and finally

PowerShell 2.0 introduced `try`, `catch`, and `finally` as a means of handling terminating errors.

`try` cannot be used alone; it must be used with either `catch`, `finally`, or both.

The most used statement is a `try` followed by a single `catch`:

```
try {
    throw 'An error'
} catch {
    Write-Host 'Caught an error'
}
```

The `catch` block only executes if an error is raised; a terminating error will stop the script enclosed by `try` at the point the error is thrown.

Inside the `catch` block, the variable `$_` (or `$PSItem`) is set to the `ErrorRecord` that caused `catch` to trigger. This is important if a command such as `ForEach-Object` is used; the original pipeline variable is not accessible inside the `catch` block.

The `ErrorRecord` has an `Exception` property, and the `Exception` has a `Message` property. These values were seen when looking at the output from `Get-Error`. These values, and any other properties of the `ErrorRecord`, may be accessed using the `$_` variable as the following code shows:

```
try {
    1/0
} catch {
    Write-Host $_.Exception.Message
}
```

In this example, the `catch` statement reacts if any exception type is thrown. The statement is equivalent to the following:

```
try {
    1/0
} catch [Exception] {
    Write-Host $_.Exception.Message
}
```

`catch` can be used to react to a more specific exception type instead:

```
try {
    throw [ArgumentException]::new('Invalid argument')
} catch [ArgumentException] {
    Write-Host 'Caught an argument exception'
}
```

More than one `catch` statement can be supplied, allowing different reactions to different types of error:

```
try {
    throw [ArgumentException]::new('Invalid argument')
} catch [InvalidOperationException] {
    Write-Host 'Caught an invalid operation exception'
} catch [ArgumentException] {
    Write-Host 'Caught an argument exception'
}
```

The `catch` statements are evaluated in the order they are written. The most specific handlers should be written before more general error types:

```
try {
    throw [ArgumentException]::new('Invalid argument')
} catch [ArgumentException] {
    Write-Host 'Caught an argument exception'
} catch {
    Write-Host 'Something else went wrong'
}
```

Each `catch` statement can be triggered by one or more exception types:

```
try {
    throw [ArgumentException]::new('Invalid argument')
} catch [InvalidOperationException], [ArgumentException] {
    Write-Host 'Argument or InvalidOperation exception'
}
```

The `finally` block and be used either alongside or instead of `catch`. The `finally` block always executes, even if an error occurred during `try`. The `finally` block is therefore an ideal place to clean up anything that might otherwise cause a problem.

In the following example, `finally` is used to close a potentially open connection to an SQL server:

```
using namespace System.Data.SqlClient

$connectionString = 'Data Source=dbServer;Initial Catalog=dbName'
try {
    $sqlConnection = [SqlConnection]::new($connectionString)
    $sqlConnection.Open()
    $sqlCommand = $sqlConnection.CreateCommand()
    $sqlCommand.CommandText = 'SELECT * FROM Employee'
    $reader = $sqlCommand.ExecuteReader()
} finally {
    if ($sqlConnection.State -eq 'Open') {
        $sqlConnection.Close()
    }
}
```

When `catch` is used with `finally`, the content of `finally` is executed before errors are returned, but after the body of `catch` has executed. This is demonstrated by the following example:

```
try {
    Write-Host "Try"
    throw 'Error'
} catch {
    Write-Host "Catch, after Try"
    throw
} finally {
    Write-Host "Finally, after Catch, before the exception"
}
```

An error raised in `try` can be repeated in a `catch` block.

Rethrowing errors

Errors may be caught using `try`, then thrown again (or rethrown) in a `catch` block. This technique can be useful if a `try` block performs several dependent steps in a sequence where one or more might fail.

Rethrowing an error raised by a script can be as simple as using `throw` in a `catch` block:

```
try {
    'Statement1'
    throw 'Statement2'
    'Statement3'
} catch {
    throw
}
```

The previous example will display `Statement1`, then the error. `Statement3` will never run in this case.

You can use `ThrowTerminatingError` to emit the error record that `catch` is handling:

```
function Invoke-Something {
    [CmdletBinding()]
    param ( )
    try {
        'Statement1'
        throw 'Statement2'
        'Statement3'
    } catch {
        $pscmdlet.ThrowTerminatingError($_)
    }
}
```

When an error is rethrown in the manner shown above, the second instance of the error (within the `catch` block) is not written to either `$Error` or a user-defined error variable. If the error is not modified, this is not a problem.

If information is added to the error before it is rethrown, such as an error ID, the modified error record will not be available to error variables. This is shown in the following example:

```
try {
    throw 'Error'
} catch {
    $params = @{
        Exception = $_.Exception
        ErrorID   = 'SomeErrorID'
        Category  = 'InvalidOperation'
    }
    Write-Error @params
}
```

`Get-Error` can be used to show that the category and `ErrorID` are not set as defined in the `catch` block for the last error raised:

```
PS> Get-Error |
>>     Select-Object CategoryInfo, FullyQualifiedErrorId |
>>     Format-List
CategoryInfo          : OperationStopped: (Error:String) [],
                        RuntimeException
FullyQualifiedErrorId : Error
```

The preceding problem can be resolved by creating a new error record with the original exception as an inner exception:

```
try {
    throw 'Error'
} catch {
    $params = @{
        Exception = [InvalidOperationException]::new(
            $_.Exception.Message,
            $_.Exception
        )
        ErrorID   = 'SomeErrorID'
        Category  = 'InvalidOperation'
    }
    Write-Error @params
}
```

In the case of the preceding exception and most exception types, the first argument of the constructor is a message, and the second (optional) argument is an inner exception.

Inner exception types cannot be used with `catch` statements in PowerShell except when a `MethodInvocationException` is raised.

A `MethodInvocationException` is a generic error that is raised when PowerShell executes a method on a .NET type or object:

```
[DateTime]::DaysInMonth(2019, 13)
Get-Error | ForEach-Object {
    $_.Exception.GetType().Name
}
```

PowerShell can react to the `MethodInvocationException` type:

```
using namespace System.Management.Automation
try {
    [DateTime]::DaysInMonth(2019, 13)
} catch [MethodInvocationException] {
    Write-Host 'Caught a method invocation exception'
}
```

However, this exception type is not particularly useful. In the case of a `MethodInvocationException` PowerShell can catch the inner exception, in this case an `ArgumentOutOfRange` exception:

```
try {
    [DateTime]::DaysInMonth(2019, 13)
} catch [ArgumentOutOfRangeException] {
    Write-Host 'Out of range'
}
```

When a command raises an error, the actual problem may be nested under one or more inner exceptions. When only the inner exception is interesting, you can use `InnerException`. It allows access to each inner exception in turn. In the following example, the property is used to access the exception in the middle:

```
try {
    throw [InvalidOperationException]::new(
        'OuterException',
        [ArgumentException]::new(
            'IntermediateException',
            [UnauthorizedAccessException]::new('InnerException')
        )
    )
} catch {
    Write-Host $_.Exception.InnerException.Message
}
```

The `Exception` class (and all derived classes) include a `GetBaseException` method. This method provides simple access to the inner-most exception and is useful when the number of nested exceptions is unknown:

```
try {
    throw [InvalidOperationException]::new(
        'OuterException',
        [ArgumentException]::new(
            'IntermediateException',
            [UnauthorizedAccessException]::new('InnerException')
        )
    )
} catch {
    Write-Host $_.Exception.GetBaseException().Message
}
```

In cases where the exception type is not sufficient, the `catch` block can be expanded to include `if` or `switch` statements, allowing decisions to be made based on values like the message in an exception.

At the start of this chapter, a reference was made to incorrect documentation surrounding terminating errors.

Inconsistent error handling

The different methods PowerShell exposes to create a terminating error are not consistent and can be extremely confusing.

The behavior of the different types of error is explored in detail in an issue in the PowerShell Docs repository: https://github.com/MicrosoftDocs/PowerShell-Docs/issues/1583.

Unfortunately, the content of the thread is yet to find a home in released documents.

When `throw` is used to raise a terminating error, the current script, and anything that called the **current script**, is stopped. The preceding link refers to errors raised by `throw` as a script-terminating error.

A script-terminating error is shown in the following example:

```
$ErrorActionPreference = 'Continue'
function caller {
    first
    second
}
function first {
    throw 'Failed'
    'first'
}
function second {
    'second'
}
```

The error raised in the function named `first` stops all commands in the chain. The function named `second` is never executed. This is shown when the function caller is run:

```
PS> caller
Exception:
Line |
   2 |      throw 'Failed'
     |      ~~~~~~~~~~~~~~
     | Failed
```

This differs from the behavior when `ThrowTerminatingError` is called. `ThrowTermatingError` is used by binary modules that want to stop execution; this introduces a rift in error handling between script and binary modules.

The article linked at the beginning of this section refers to an error raised by `ThrowTerminatingError` as a statement-terminating error:

```
using namespace System.Management.Automation
function caller {
    first
    second
}
function first {
    [CmdletBinding()]
    param ( )

    $errorRecord = [ErrorRecord]::new(
        [Exception]::new('Failed'),
        'ID',
        'OperationStopped',
        $null
    )
    $pscmdlet.ThrowTerminatingError($errorRecord)
    'first'
}
function second {
    'second'
}
```

Running the function first shows a different result to the version that used a `throw` statement instead:

```
PS> caller
first:
Line |
   2 |      first
     |      ~~~~~
     | Failed
Second
```

The `ThrowTerminatingError` statement stops the function `first` from completing, but it does not stop the function `caller` from continuing.

This second version is therefore consistent with a script that runs a command from a binary module. In the following example, `ConvertFrom-Json` raises a terminating error but does not stop the function that called it from executing:

```
function caller {
    ConvertFrom-Json -InputObject '{{'
    second
}
function second {
    'second'
}
```

The output when running `caller` is:

```
PS> caller
ConvertFrom-Json:
Line |
   2 |      ConvertFrom-Json -InputObject '{{'
     |      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     | Conversion from JSON failed with error: Invalid property identifier character: {. Path '', line
1, position 1.
second
```

The same behavior is seen when calling .NET methods. The static method, `IPAddress.Parse`, will raise an exception because the use of the method is not valid. The function continues from this error and calls the function `second`:

```
function caller {
    [IPAddress]::Parse('this is not an IP')
    child1
}
function second {
    'second'
}
caller
```

Not only do errors raised by throw differ from those raised by `ThrowTerminatingError`, but `throw` statements are also influenced by the error action preference despite the documentation for PowerShell having stated the opposite for the last decade.

throw and ErrorAction

The `throw` keyword raises a terminating error; terminating errors are described in `about_Preference_Variables` as not being affected by the `ErrorActionPreference`.

Unfortunately, errors raised by throw are affected by `ErrorAction` when `ErrorAction` is set to `SilentlyContinue`. This behavior is an important consideration when designing commands for others to use.

The following function throws a terminating error; the second statement should never run:

```
function Invoke-Something {
    [CmdletBinding()]
    param ( )

    throw 'Error'
    Write-Host 'No error'
}
```

Running the function with an error action set to the default, `continue`, shows that the error is thrown and the second command does not execute:

```
PS> Invoke-Something
Exception:
Line |
   5 |      throw 'Error'
     |      ~~~~~~~~~~~~~
     | Error
```

If `ErrorAction` is set to `SilentlyContinue`, `throw` will be ignored:

```
PS> Invoke-Something -ErrorAction SilentlyContinue
No error
```

The same problem exhibits of the `$ErrorActionPreference` variable is set in the parent scope:

```
PS> $ErrorActionPreference = 'SilentlyContinue'
PS> Invoke-Something
No error
```

You can still use a `throw` statement in a script, but because it does not behave as described in the documentation it must be used with great care.

Enclosing a `throw` statement in a `try` block will cause it to trigger `catch`, ending the script as it should regardless of the `ErrorAction` setting:

```
function Invoke-Something {
    [CmdletBinding()]
    param ( )
    try {
        throw 'Error'
        Write-Host 'No error'
    } catch {
        Write-Host 'An error occurred'
    }
}
```

When the function is called, the content of the `catch` block executes as it should:

```
PS> Invoke-Something -ErrorAction SilentlyContinue
An error occurred
```

The problem described here also applies when `throw` is used within the `catch` block, although, in this case, the script still terminates.

The following script should result in an error being displayed as the error is terminating; however, no error is displayed if you set `-ErrorAction` to `SilentlyContinue`. The error raised in `try` does still prevent the script from progressing to the `Write-Host` command:

```
function Invoke-Something {
    [CmdletBinding()]
    param ( )
    try {
        throw 'Error'
        Write-Host 'No error'
    } catch {
        throw 'An error occurred'
    }
}
```

In this version, any statements that appear after the `throw` statement in the `catch` block will not execute. When `-ErrorAction` is set to `SilentlyContinue`, this function runs and returns nothing at all. The only evidence of an error is in the `$Error` variable:

```
PS> $Error.Clear()
PS> Invoke-Something -ErrorAction SilentlyContinue
PS> $Error[0]
Exception:
Line |
   9 |          throw 'An error occurred'
     |          ~~~~~~~~~~~~~~~~~~~~~~~~~
     | An error occurred
```

For scripts that use the `CmdletBinding` attribute, `ThrowTerminatingError` can be used. The `ThrowTerminatingError` method is not affected by the `-ErrorAction` parameter or `$ErrorActionPreference` variable:

```
function Invoke-Something {
    [CmdletBinding()]
    param ( )
    try {
        throw 'Error'
        Write-Host 'No error'
    } catch {
        $PSCmdlet.ThrowTerminatingError($_)
    }
}
```

Running the command shows will reliably show the error. The error can be handled by using a `try` statement in the script calling this command:

```
PS> Invoke-Something -ErrorAction SilentlyContinue
Exception: Error
```

To establish consistent behavior, the following recommendations can be made:

- Prefer to use `CmdletBinding` with functions, scripts, and `ScriptBlocks`
- When using `throw`, only use `throw` inside `try` block
- Use the `ThrowTerminatingError` method to stop a script from executing
- Prefer the `WriteError` method when creating non-terminating errors as this correctly sets the value for `$?`
- Any script that calls another that may raise an error should expect and handle errors using a `try` block

To provide more granular error handling in a script, you can nest `try` blocks.

Nesting try, catch, and finally

One `try` statement can be inside another to provide granular error handling of small operations.

A script that performs setup actions and then works on several objects in a loop is a good example of a script that might benefit from more than one `try` statement. The script should terminate cleanly if something goes wrong during setup, but it might only notify if an error occurs within the loop.

The following functions can be used as a working example of such a script. The setup actions might include connecting to a management server or a data source of some kind:

```
function Connect-Server {}
```

Once the connection is established, a set of objects might be retrieved:

```
function Get-ManagementObject {
    1..10 | ForEach-Object {
        [PSCustomObject]@{
            Name     = $_
            Property = "Value$_"
        }
    }
}
```

These objects might be modified by another function, which occasionally goes wrong:

```
function Set-ManagementObject {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory, ValueFromPipeline)]
        $Object,

        $Property
    )
    process {
        try {
            if (Get-Random -Maximum 2) {
                throw 'something went wrong!'
            }
            $Object.Property = $Property
        } catch {
            $PSCmdlet.ThrowTerminatingError($_)
        }
    }
}
```

The following script uses the previous functions. If a terminating error is raised during either the `Connect` or `Get` commands, the script will stop. If a terminating error is raised when executing the `Set` command, the script writes a non-terminating error and moves onto the next object:

```
try {
    Connect-Server
    Get-ManagementObject | ForEach-Object {
        try {
            $_ | Set-ManagementObject -Property 'NewValue'
        } catch {
            Write-Error -ErrorRecord $_
        }
    }
} catch {
    throw
}
```

A `throw` is used in the `catch` block of the statement above as it does not include a `CmdletBinding` attribute and a `param` block.

Before `try`, `catch`, and `finally` were introduced handling exceptions in PowerShell required the use of a `trap` statement.

About trap

All the way back in PowerShell 1.0, a `trap` statement was the only way to handle terminating errors in a script.

As `trap` very occasionally finds its way into modern scripts, it is beneficial to understand the use of the statement.

`trap` is used to catch errors raised anywhere within the scope of the `trap` declaration, that is, the current scope and any child scopes.

The PowerShell engine finds `trap` statements anywhere within a script before beginning execution of the script. It deviates from the line-by-line approach one might expect when running a PowerShell script.

Using trap

A `trap` statement is declared in a similar manner to the `catch` block. A `trap` statement can be created to handle any exception type:

```
trap {
    Write-Host 'An error occurred'
}
```

A `trap` statement can also be created to handle a specific exception type:

```
trap [ArgumentException] {
    Write-Host 'Argument exception'
}
```

A script may contain more than one `trap` statement; for example:

```
trap [InvalidOperationException] {
    Write-Host 'An invalid operation'
}
trap {
    Write-Host 'Catch all other exceptions'
}
```

The ordering of the preceding `trap` statements does not matter; the statement with the most specific error type is always used to handle a given error.

The following example uses a `trap` statement at the end of the script to illustrate that the location of the statement is unimportant:

```
& {
    Write-Host 'Statement1'
    throw 'Statement2'
    Write-Host 'Statement3'

    trap { Write-Host 'An error occurred' }
}
```

The error raised by `throw` causes the `trap` statement to execute, and then execution stops; `Statement3` is never written.

trap, scope, and continue

By default, if an error is handled by `trap`, script execution stops.

You can use the `continue` keyword to resume a script at the next statement.

The following example handles the error raised by `throw` and continues onto the next statement:

```
& {
    Write-Host 'Statement1'
    throw 'Statement2'
    Write-Host 'Statement3'

    trap {
        Write-Host 'An error occurred'
        continue
    }
}
```

The behavior of `continue` is dependent on the scope the `trap` statement is written in. In the preceding example, `continue` moves onto writing `Statement3` as the `trap` statement, and the statements being executed are in the same scope.

The following script declares a function that throws an error. `trap` is declared in the parent scope of the function:

```
& {
    function Invoke-Something {
        Write-Host 'Statement1'
        throw 'Statement2'
        Write-Host 'Statement3'
    }

    Invoke-Something
    Write-Host 'Done'

    trap {
        Write-Host 'An error occurred'
        continue
    }
}
```

The `continue` keyword is used, but `Statement3` is not displayed. Execution can only continue in the same scope as the `trap` statement.

Summary

Error handling in PowerShell is a complex topic, perhaps made more so by inconsistencies in the documentation that can trip up new and experienced PowerShell users.

PowerShell includes the concept of non-terminating errors. Non-terminating errors allow a script, such as one acting on a pipeline, to carry on in the event of an error.

Error handling cannot be said to be consistently implemented in PowerShell. It is not always true that a problem that prevents an action from continuing will be described as a terminating error. The reverse is also true: not all actions that allow an action to continue are described as non-terminating errors. Great care must therefore be taken when writing code to correctly handle error conditions.

Non-terminating errors should be used when writing commands that expect to act on more than one object if the error is restricted to that one object and does not prevent a broader activity from completing.

You use terminating errors when execution absolutely cannot continue.

PowerShell includes several different ways to raise both terminating and non-terminating errors. `Write-Error`, `$PSCmdlet.WriteError`, `throw`, and `$PSCmdlet.ThrowTerminatingError` were all introduced in this chapter.

The problems associated with the use of `throw` were explored, and there were general recommendations that `throw` should be confined to `try` and `$PSCmdlet.ThrowTerminatingError` should be used to end a script in the event of a terminal problem.

Finally, `trap` statements were demonstrated. `trap` came along with PowerShell 1 and is rarely used in modern scripts. `trap` was largely superseded by `try`, `catch`, and `finally`, which were introduced with PowerShell 2.

The next chapter explores the debugging options available in PowerShell.

23

Debugging and Troubleshooting

Debugging is the art of discovering, isolating, and fixing bugs in code. The complexity of debugging increases with the complexity of the code.

To a degree, the need to debug complex scripts can be reduced by adopting certain development strategies:

- Scripts should be broken down into discreet units by using functions
- Each unit of code should strive to excel at one thing only
- Each unit of code should strive to be as short as it can reasonably be without sacrificing clarity
- Tests should be developed to ensure that each unit of code acts as it should
- Use commands such as `Write-Verbose` or `Write-Debug` to write identifiable messages, perhaps including values of variables

Debugging is an inevitable part of the development process. Fortunately, the debugger itself is not difficult to use.

This chapter explores the following topics:

- Common problems
- Debugging in the console
- Debugging in Visual Studio Code

Before looking at specific debugging tools, some of the more common problems can be explored.

Common problems

When supporting PowerShell development, some problems appear again and again. No one is immune from writing a bug into code, no matter how experienced. All experience brings is the ability to find and fix bugs more quickly.

This section explores the following relatively common problems:

- Dash characters
- Operator usage
- Use of named blocks
- Problems with variables

The dash character is a relatively common problem when a piece of code is copied from a blog article or when any code has been corrected by a word processor.

Dash characters

In PowerShell, a hyphen is used to separate the verb from the noun in command names and is also used to denote a parameter name after a command.

When looking for examples on the internet, it is common to bump into PowerShell code that has been formatted into rich text. That is, where the hyphen character has been replaced by a dash character, such as an em or en dash.

In printed text, this problem is difficult to show, which is why it is a problem in the first place.

The command may appear ordinary:

```
Get-Process | Where-Object WorkingSet64 -gt 100MB
```

If you paste it into the PowerShell console, the dash characters disappear and PowerShell will complain that the command does not exist:

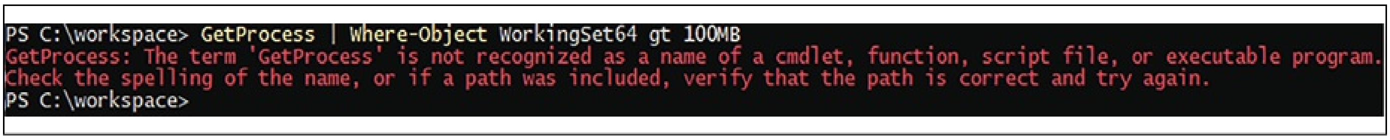


Figure 23.1: Pasting into a console

If you run the command in either ISE (under PowerShell 5.1) or the Visual Studio integrated terminal, the dash character stays, but PowerShell still raises an error about the command name. This error message is even less useful because the dash is still present, and the command name appears correct:

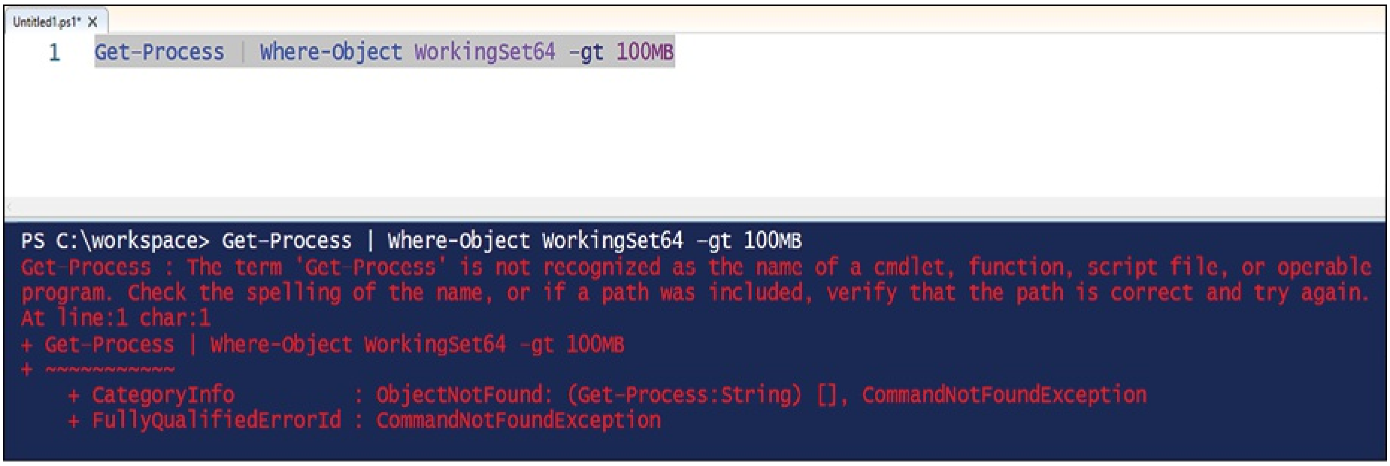


Figure 23.2: Error when running a selection in ISE

The same error shows in Visual Studio Code. This time, the code is executed in PowerShell 7.1.3:

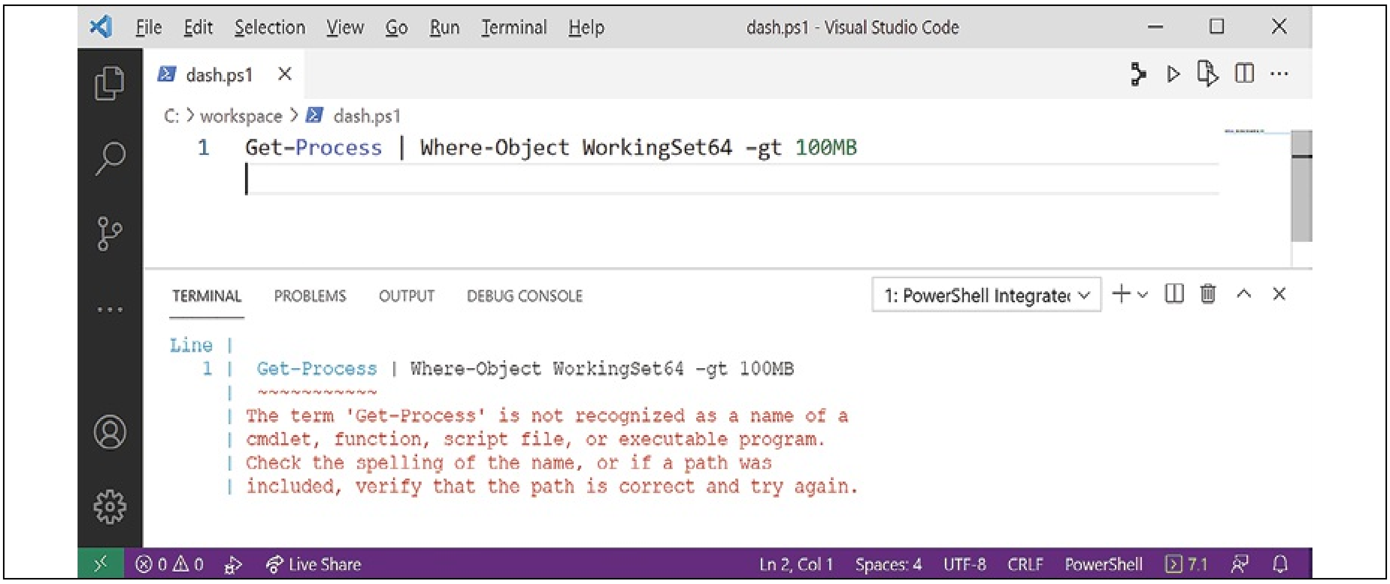


Figure 23.3: Error when running a selection in VS Code

The syntax highlighter in Visual Studio Code hints at the problem. The `Get-Process` command in the script has two different colors (the actual colors will depend on which themes are enabled). The `Get-Process` command should have the same highlighting as the `Where-Object` command in this example.

The color difference is because the dash character used is not a hyphen, it is an en dash, and this confuses PowerShell new and old.

Fixing the dash in the `Get-Process` command (by retyping the character) fixes the `Get-Process` command, but despite the

filter appearing to be correct, the filter will still fail to apply.

The same dash appears again in the `-gt` parameter, and it must also be fixed for the code to work as intended.

If the problem described previously does not show when pasting the preceding example, the example can be recreated from the following string:

```
"Get$([char]8211)Process | Where-Object WorkingSet64 $([char]8211)gt 100MB"
```

It takes a sharp eye to spot a problem based only on a tiny difference in the length of a dash character. Syntax highlighting helps a lot, but this does not clearly point out that there is a problem.

> **PSScriptAnalyzer**
>
> The default rules in `PSScriptAnalyzer` do not warn about this problem, but it is possible to create a rule. My module, `Indented.ScriptAnalyzerRules`, includes a rule for this problem, `AvoidDashCharacters`: https://github.com/indented-automation/Indented.ScriptAnalyzerRules.

Operators in PowerShell are another opportunity for bugs to creep into code.

Operator usage

Bugs in code can easily be caused by the incorrect usage of an operator. This can be a logical error, such as using `-gt`, where `-ge` was a better choice (or vice versa).

Operators were explored in detail in *Chapter 4, Operators*.

Problems can also be caused if `=` is accidentally used instead of `-eq`.

Assignment instead of equality

Accidentally using the assignment operator, `=`, in place of the equality operator, `-eq`, is a common problem.

When an assignment operation is enclosed in brackets, PowerShell performs the assignment and returns the value that was assigned, for example:

```
PS> ($variable = $true)
True
```

The same applies if the assignment statement is part of an `if` statement:

```
if ($variable = $true) {
    Write-Host 'The condition is true'
} else {
    Write-Host 'The condition is false'
}
```

In this example, the variable will always have the value `$true` because of the assignment inside the `if` statement.

PowerShell will not highlight this assignment as being an error. It is a legitimate use of the assignment operator. The following example captures the output from the `Get-Process` command at the same time as testing it:

```
$params = @{
    Name        = 'notepad'
    ErrorAction = 'SilentlyContinue'
}
if ($process = Get-Process @params) {
    Write-Host "$($process.Name) is running, ID is $($process.ID)"
} else {
    Write-Host "$($params.name) is not running"
}
```

It would be easy to argue that this is not a great practice. However, there is a difference between what is possible and what might be considered a good practice. Since PowerShell will not warn you when this feature is erroneously used, it is important to be mindful of attempts to compare values in this manner.

Comparison operators are not the only type of operator that can occasionally cause confusion.

-or instead of -and

The logical operators `-and` and `-or` are occasionally easy to confuse. This typically exhibits when converting a thought in a natural language directly into code.

For example, the following statement can be translated into code:

```
I want all names which do not start with a or b.
```

When writing the code to implement the statement, it is tempting to use the same logical expression as used in natural language; in this case, that is "or":

```
@(
    'Anna'
    'Ben'
    'Chris'
    'David'
) | Where-Object { $_ -notlike 'a*' -or $_ -notlike 'b*' }
```

Each comparison is evaluated without considering the other. The preceding code will return `Anna`, because `Anna` does not start with the letter "b", and it will return `Ben`, because `Ben` does not begin with the letter "a".

Using `-and` in place of `-or` will fix the problem in the code, as shown here:

```
@(
    'Anna'
    'Ben'
    'Chris'
    'David'
) | Where-Object { $_ -notlike 'a*' -and $_ -notlike 'b*' }
```

A less common problem can be seen when using a comparison operator on an array.

Negated array comparisons

Each of the comparison operators includes a negated version, that is, `-eq` and `-ne`, `-match` and `-notmatch`, and so on.

When performing a comparison on an array, it can be tempting to use the negative version of the operator to make an assertion.

In this example, the `-notmatch` operator is erroneously used to assert that there are no names in the array that start with " `a` " or " `b` ":

```
$array = @(
    'Anna'
    'Ben'
    'Chris'
    'David'
)
if ($array -notmatch '^[ab]') {
    Write-Host "No names starting A or B"
}
```

When acting on an array, the comparison operator does not return a simple `$true` or `$false`; it returns the elements in the array that satisfy the comparison. In the preceding example, the result of the comparison is the strings `Chris` and `David`.

If you use the `-not` operator in conjunction with the `-match` operator instead, the statement in the body of the `if` statement becomes more accurate:

```
$array = @(
    'Anna'
    'Ben'
    'Chris'
    'David'
)
if (-not ($array -match '^[ab]')) {
    Write-Host "No names starting A or B"
}
```

It is easy to make a mistake with a comparison operator and, at times, it can be difficult to spot such problems. Testing is an important part of any development process.

PowerShell uses named blocks to support pipeline operations.

Use of named blocks

The named blocks `begin`, `process`, and `end` are used to support pipeline operations in PowerShell. Named blocks are also used if dynamic parameters are in use.

Named blocks are explored in *Chapter 17, Scripts, Functions, and Script Blocks*.

Dynamic parameters are explored in *Chapter 18, Parameters, Validation, and Dynamic Parameters*.

When a named block is used, all code must be placed inside named blocks.

Code outside of a named block

Placing code outside of a named block when named blocks are in use can result in several different errors.

In the following example, a verbose statement has erroneously been placed before the `process` block. The function uses a brace style where the opening brace is placed on the line after the statement:

```
function Get-Something
{
    [CmdletBinding()]
    param
    (
        [Parameter(ValueFromPipeline)]
        [string]$InputObject
    )
    Write-Verbose 'Starting Get-Something'
code
    process
    {
        Write-Verbose "Working on $InputObject"
    }
}
```

It would be easy to expect an editor to regard this script as invalid; however, it is not. When it is run, the following happens:

1. The `Write-Verbose` statement will run (showing a message using `-Verbose`)
2. A list of processes will be displayed
3. The script block containing the second `Write-Verbose` statement will be emitted, but not executed

These actions happen because the first statement after `param` defines the block that is used. `Write-Verbose` means all content is placed in the `end` block (the default for the function).

The `process` statement is considered a command and implicitly aliased to `Get-Process`. PowerShell attempts to find and run a `Get-` command for all bare words that do not explicitly resolve to another command; for example, running `service` will cause `Get-Service` to execute.

If the braces were placed on the same line as the preceding statement, the outcome would be slightly different:

```
function Get-Something {
    [CmdletBinding()]
    param (
        [Parameter(ValueFromPipeline)]
        [string]$InputObject
    )
    Write-Verbose 'Starting Get-Something'
    process {
        Write-Verbose "Working on $InputObject"
    }
}
```

This time, when the function is run, the output is as follows:

1. The `Write-Verbose` statement will run (showing a message using `-Verbose`)
2. `Get-Process` attempts to run, but an error is thrown because the script block containing `Write-Verbose` is not a valid parameter

The error is:

```
PS> Get-Something
Get-Process:
Line |
  10 |      process {
     |              ~
     | Cannot evaluate parameter 'Name' because its argument is specified as a script block and there
is no input. A script block cannot be evaluated without input.
```

When one named block is used, all code must be placed inside a named block. Moving the initial `Write-Verbose` statement into a `begin` block will fix the problem with the function.

If no blocks are defined, PowerShell uses the default block. For functions, scripts, and script blocks, the default block is `end`. For functions created using the `filter` keyword, the default block is `process`.

Pipeline without process

You can create functions with parameters that accept pipeline input. If the body of the function is not placed inside a named block, then PowerShell will use the default block, `end`.

When you add a pipeline parameter to a function that only implements an `end` block, the body of the function will only ever be able to use the last value in the pipeline:

```
function Write-Number {
    [CmdletBinding()]
    param (
        [Parameter(ValueFromPipeline)]
        [int]$Number
    )
    Write-Host $Number
}
```

When running in a pipeline, only the last of the input values will be displayed:

```
PS> 1..5 | Write-Number
5
```

Moving the body of the function into a `process` block will allow it to act on all values from the input pipeline:

```
function Write-Number {
    [CmdletBinding()]
    param (
        [Parameter(ValueFromPipeline)]
        [int]$Number
    )
    process {
        Write-Host $Number
    }
}
```

Perhaps one of the most obvious causes of bugs is a problem with a variable.

Problems with variables

Variables are a critical part of almost every script. Potential problems with variables include:

- Typing errors
- Incorrectly assigned types
- Accidental use of reserved variables

It is difficult to solve the problems above using tools; context plays a large part in determining what is correct. Understanding context often requires a human actor. Humans are particularly good at spotting patterns, while computers must be taught based on large sample sets.

One possible technical solution you can use to reduce the risk impact of typing errors is strict mode in PowerShell.

About strict mode

Strict mode in PowerShell is used to add several additional parser rules when evaluating scripts. Strict mode is enabled using the `Set-StrictMode` command. The mode is applied to the current scope and all child scopes.

`Set-StrictMode` can be used in a module without affecting global scope or any other modules.

You can set strict mode to three (effective) values:

- 1.0
- 2.0
- Latest

For example, the following command sets strict mode to `Latest`:

```
Set-StrictMode -Version Latest
```

The effect of each of these modes is described in the help file for `Set-StrictMode`. When you use `Latest` as the mode, the effect in PowerShell is that it:

- Prohibits the use of uninitialized variables
- Prohibits references to non-existent properties of objects
- Prohibits function calls that use the syntax for calling methods
- Prohibits out of bounds or unresolvable array indexes

For example, when enabled in the following function, strict mode causes an error because of the mistake in a variable name:

```
function Test-StrictMode {
    Set-StrictMode -Version Latest
    $names = 'pwsh', 'powershell'
    foreach ($name in $naems) {
        Write-Host $name
    }
}
```

When the function runs, an error relating to the variable name is displayed:

```
PS> Test-StrictMode
InvalidOperation:
Line |
   5 |      foreach ($name in $naems) {
     |                        ~~~~~~
     | The variable '$naems' cannot be retrieved because it has not been set.
```

It is important to note that `Set-StrictMode` does not enforce scope, so if, for some reason, the `$naems` variable were to exist in a parent scope, the error would not be displayed.

`Set-StrictMode` has a mixed reputation in the PowerShell community. On the one hand, it offers some small protection against errors. On the other hand, it prohibits certain simple patterns, for instance, testing for a non-existent property on an object.

Without strict mode, it is possible to test for a null, empty, false, or non-existent property in an `if` statement, as the following shows:

```
$object = [PSCustomObject]@{
    ValueA = 1
}
if ($object.ValueB) {
    Write-Host "ValueB is set"
}
```

Enabling strict mode will instead show an error:

```
PS> Set-StrictMode -Version Latest
PS>if ($object.ValueB) {
>>     Write-Host "ValueB is set"
>> }
PropertyNotFoundException: The property 'ValueB' cannot be found on this object. Verify that the
property exists.
```

To accommodate strict mode, the condition must become much more complex, making use of the hidden `PSObject` member of objects in PowerShell:

```
Set-StrictMode -Version Latest
$object = [PSCustomObject]@{
    ValueA = 1
}
if ($object.PSObject.Properties.Item('ValueB') -and
    $object.ValueB) {
    Write-Host "ValueB is set"
}
```

Strict mode can be disabled at any time by using the following command:

```
Set-StrictMode -Off
```

Perhaps most important of all is the fact that the protection strict mode brings is displayed at runtime. As a developer, the problem with the variable ideally needs to be shown in an editor when writing code. By the time a script is handed to someone else to run, it is far too late to show warnings about variable usage. It is up to each developer to decide whether strict mode is appropriate.

Beyond typing mistakes, one of the most common errors with variables is caused by assigning a type to a variable.

Variables and types

Variables in PowerShell may be assigned a type on creation by placing the type on the left-hand side of the variable. For example, the variable in the following code is defined as having a `string` type:

```
[string]$string = 'Hello world'
```

Any subsequent assignment to the variable will be coerced into a string:

```
PS> $string = @{}
PS> $string
System.Collections.Hashtable
PS> $string.GetType()
IsPublic IsSerial Name          BaseType
-------- -------- ----          --------
True     True     String        System.Object
```

The type applies to all assignments made to that variable until either another type is assigned (on the left-hand side of the variable) or the variable is destroyed using the `Remove-Variable` command.

This is most commonly a problem where a parameter for a command is created with a type, and an attempt is made to reuse the variable in the body script or function.

Assigning a type to a variable can have interesting consequences when applied to an automatic variable.

Types and reserved variables

PowerShell includes many built-in variables; these variables are described in the `about_automatic_variables` help file:

```
Get-Help about_automatic_variables
```

Several of these variables only exist in a specific context. For example, the `$foreach` variable only exists inside a `foreach` loop.

If you accidentally use a reserved variable, for instance, as a parameter, and the parameter has an associated type, this can break functionality in PowerShell.

For example, the following `switch` statement will act on either `true` or `false`:

```
switch ($true) {
    $true { 'The value is true' }
    $false { 'The value is false' }
}
```

If a type is erroneously assigned to a variable called `$switch`, the statement will no longer function as the iterator it requires is broken. Instead, an error will be displayed:

```
[switch]$Switch = $true
switch ($true) {
    $true { 'The value is true' }
    $false { 'The value is false' }
}
```

The following `System.SZArrayEnumerator` error is the type the `$switch` variable should have but no longer does because of the previous assignment statement:

```
MetadataError: Cannot convert value "System.SZArrayEnumerator" to type
"System.Management.Automation.SwitchParameter". Boolean parameters accept only Boolean values and
numbers, such as $True, $False, 1 or 0.
```

PowerShell will not prevent this from happening; it is up to a developer to avoid this problem in the first place.

The problem with `switch` in the preceding example will persist until either PowerShell is restarted or the variable is removed:

```
Remove-Variable switch
```

It is important to be mindful of and avoid using automatic variables in code.

Being aware of potential problems avoids the need for extensive debugging. When the problem is less obvious, the debugger can be used.

Debugging in the console

The PowerShell debugger allows code execution to be paused and the state of a script to be analyzed at a specific point.

These points are known as **breakpoints** and are set using the `Set-PSBreakpoint` command.

PowerShell describes the following operations in the `about_Debuggers` help file:

```
Get-Help about_Debuggers
```

You can use the `Set-PSBreakpoint` command to set a breakpoint when a command is run, when a variable is used, or on a specific line in a saved script.

Setting a command breakpoint

Setting a breakpoint on a command will trigger the debugger when that command is run.

In the next example, a breakpoint is created that triggers when the `Get-Process` command runs. As `Get-Process` is inside a loop, it will be possible to inspect the state of variables inside the loop in the debugger:

```
Set-PSBreakpoint -Command Get-Process
$names = 'powershell', 'pwsh', 'code'
foreach ($name in $names) {
    Get-Process $name -ErrorAction SilentlyContinue
}
```

When the example is run, the `DBG` prompt will appear:

```
Hit Command breakpoint on 'Get-Process'
At line:2 char:5
+     Get-Process $name -ErrorAction SilentlyContinue
+     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[DBG]: PS C:\workspace>>
```

Pressing `?` at the DBG prompt will show the possible debug actions. The output is shown in *Figure 23.4*:

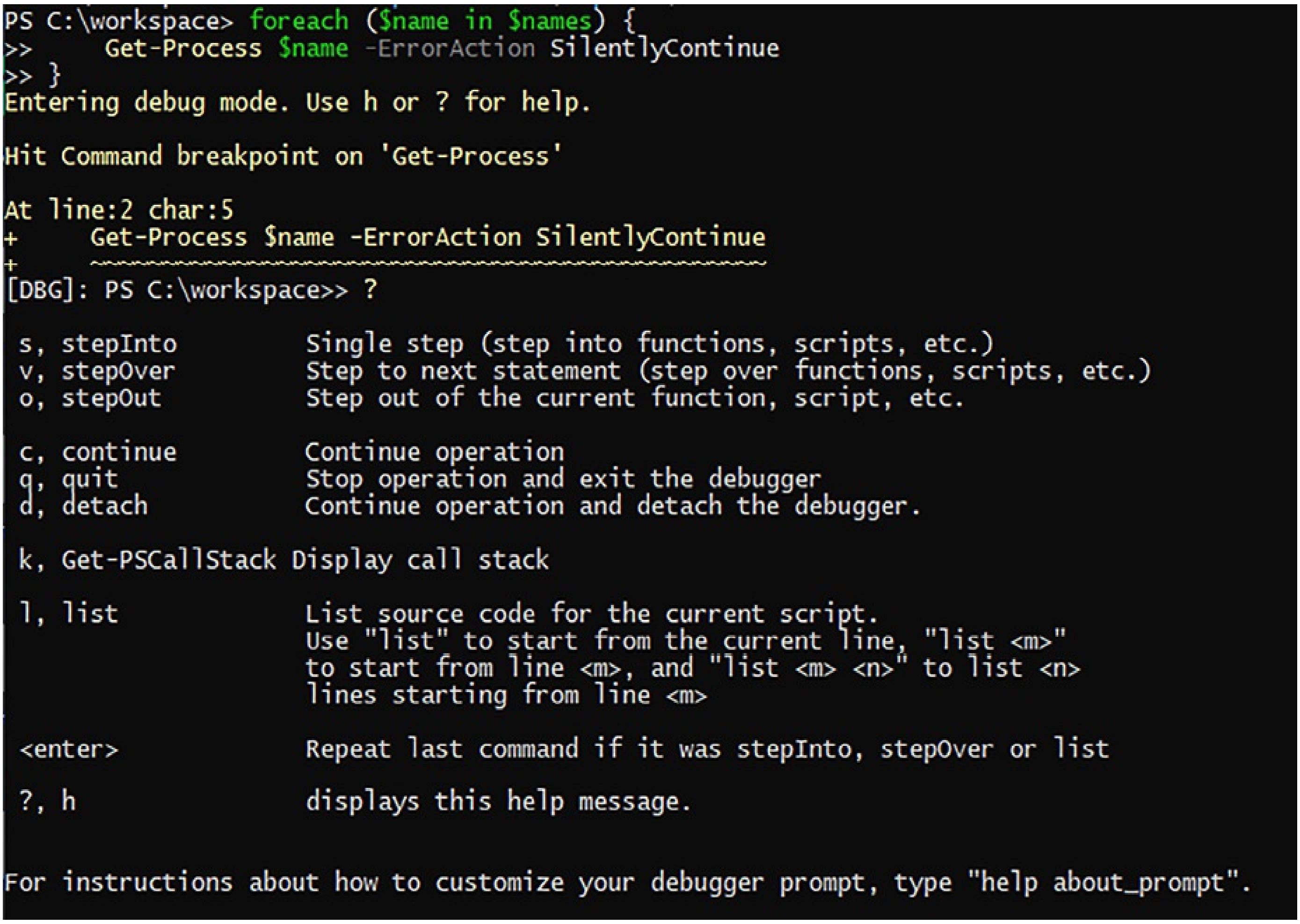


Figure 23.4: Error when running a selection in VS Code

In addition to these actions, any variable values may be inspected at this point in the loop simply by typing the variable name into the prompt:

```
[DBG]: PS C:\workspace>> $name
powershell
```

The current script may be displayed using the `list` command inside the debug prompt. By default, it will list the entire script. A range of lines may be displayed by specifying optional start and end line numbers for the list command:

```
[DBG]: PS C:\workspace>> list 1 2
    1:  foreach ($name in $names) {
    2:*     Get-Process $name -ErrorAction SilentlyContinue
```

If the end line is not specified, the command will display all script lines from the start to the end of the script.

Pressing `c`, to continue, will move to the next iteration in the loop. If any of the processes are not running, an error will be displayed by `Get-Process`. This time, the `$name` variable will show the second item in the loop.

```
[DBG]: PS C:\workspace>> $name
pwsh
```

If the process is running, the process objects will be returned. Once the script completes, the debug prompt will close.

Breakpoints in a session can be viewed using the `Get-PSBreakpoint` command, and any existing breakpoint can be removed using `Remove-PSBreakpoint`. The following command removes all breakpoints in the session:

```
Get-PSBreakpoint | Remove-PSBreakpoint
```

Confusing breakpoints

Breakpoints should be removed after running any examples to avoid confusing results.

Breakpoints may also be set on variables.

Using variable breakpoints

When a breakpoint is set on a variable, it will, by default, only trigger when the variable is set (written to).

Writing to a variable means changing the value held by the variable. In the following example, the value of the `$newValue` variable is set five times, once per iteration of the loop:

```
foreach ($value in 1..5) {
    $newValue = $value
}
```

Setting a breakpoint based on this variable will therefore cause the debugger to pause execution five times:

```
Set-PSBreakpoint -Variable newValue
foreach ($value in 1..5) {
    $newValue = $value
}
```

If a variable holds a collection, such as a Hashtable, it is important to note that adding a key is not a write operation with respect to the variable. In this example, the debugger will only trigger when `$values` is created:

```
Set-PSBreakpoint -Variable values
$values = @{}
foreach ($value in 1..5) {
    $values[$value] = $value
}
```

The operation to add a key to the Hashtable is performed on the value of the variable, which is read from the variable object.

The `-Mode` parameter, which has the default value `Write`, may be used to trigger the debugger when a variable is read by setting the argument to either `Read` or `ReadWrite`.

If the last breakpoint is removed, and a new breakpoint added, the debugger will trigger five times again:

```
Set-PSBreakpoint -Variable values -Mode Read
$values = @{}
foreach ($value in 1..5) {
    $values[$value] = $value
}
```

The most common use of the debugger is a line breakpoint.

Setting a line breakpoint

A line breakpoint can only be set if a script is saved to a file. The debugger will trigger when the line in the script is executed. The debugger may be used to explore the current state:

```
@'
$names = 'powershell', 'pwsh', 'code'
foreach ($name in $names) {
    Get-Process $name -ErrorAction SilentlyContinue
}
'@ | Set-Content script.ps1
Set-PSBreakpoint -Script script.ps1 -Line 3
.\script.ps1
```

When accessing script variables inside the debugger, several automatic variables cannot be viewed:

- `$Args`
- `$Input`
- `$MyInvocation`
- `$PSBoundParameters`

These variables are used by the debugger and are therefore overwritten by the debugger, hiding any values the script might use.

The value of an automatic variable should be assigned to another named variable in the script to view the content of that variable. For example, the `$PSBoundParameters` variable value is assigned to another variable in the following example:

```
@'
param (
    [string[]]$Name
)
$boundParameters = $PSBoundParameters
foreach ($processName in $Name) {
    Get-Process $name -ErrorAction SilentlyContinue
}
'@ | Set-Content script.ps1
Set-PSBreakpoint -Script script.ps1 -Line 7
```

When the debugger starts, you can inspect the value of the `$boundParameters` variable, but the value of `$PSBoundParameters` is not available. This is shown in the following snippet:

```
PS C:\workspace> .\script.ps1 -Name 'pwsh'
Hit Line breakpoint on 'C:\workspace\script.ps1:7'
At C:\workspace\script.ps1:7 char:5
+     Get-Process $name
+     ~~~~~~~~~~~~~~~~~
[DBG]: PS C:\workspace>> $PSBoundParameters
[DBG]: PS C:\workspace>> $boundParameters
Key  Value
---  -----
Name {pwsh}
```

Setting breakpoints using line numbers in the console is possible, but is not the easiest way to work with the debugger.

The Visual Studio Code editor includes a debugger interface, which is a lot easier to work with than the command line.

Debugging in Visual Studio Code

Visual Studio Code and other interactive editors greatly simplify working with the debugger. The debugger is accessed via a button on the left-hand side of the editor.

Using the debugger

The debugging options in Visual Studio Code, by default, will run a script and stop at any defined breakpoint.

The `param` block is removed from `script.ps1` for this example, making the content:

```
$names = 'powershell', 'pwsh', 'code'
foreach ($name in $names) {
    Get-Process $name -ErrorAction SilentlyContinue
}
```

You can add breakpoints to a script by clicking to the left of the line number. A breakpoint has been added to `script.ps1`, as shown in *Figure 23.5*:

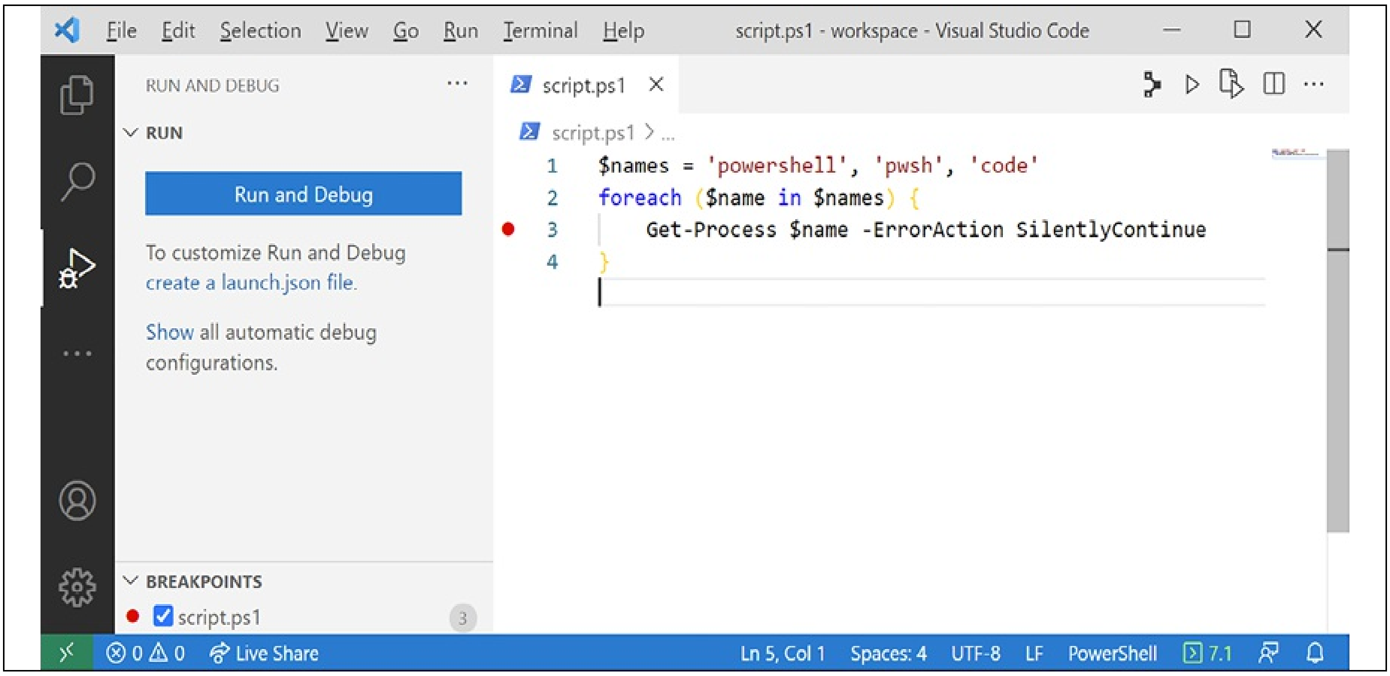


Figure 23.5: Debugging in Visual Studio Code

The breakpoint appears as a red dot next to the line. When the **Run and Debug** button is pressed, Visual Studio Code will execute the script. The script in this case is run in PowerShell 7.1 based on the version in the bottom-left corner of the editor.

When run, the **VARIABLES** and **CALL STACK** boxes will fill as shown in *Figure 23.6*:

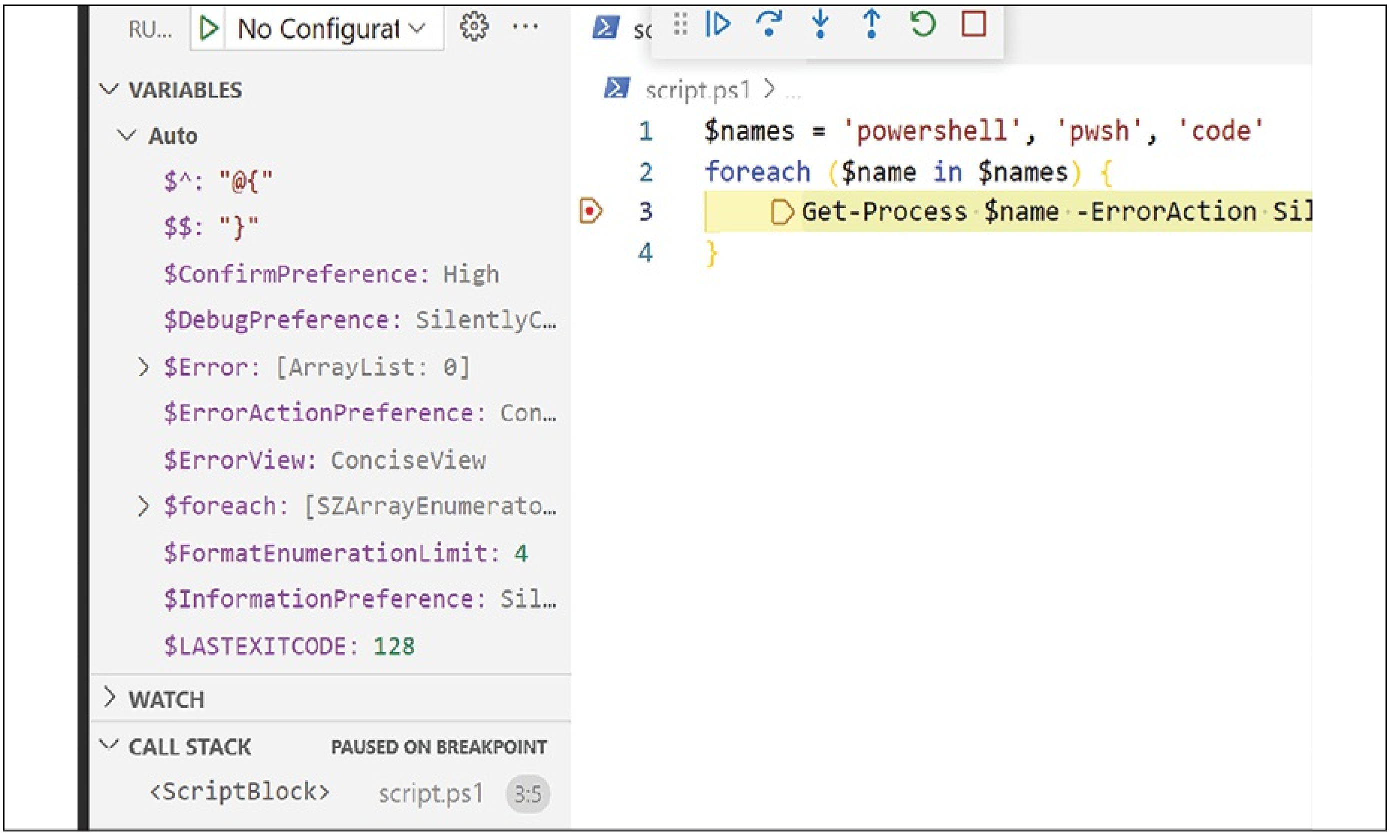


Figure 23.6: Accessible variables in the debugger

The icons in the bar at the top of the window present the different options for the debugger:

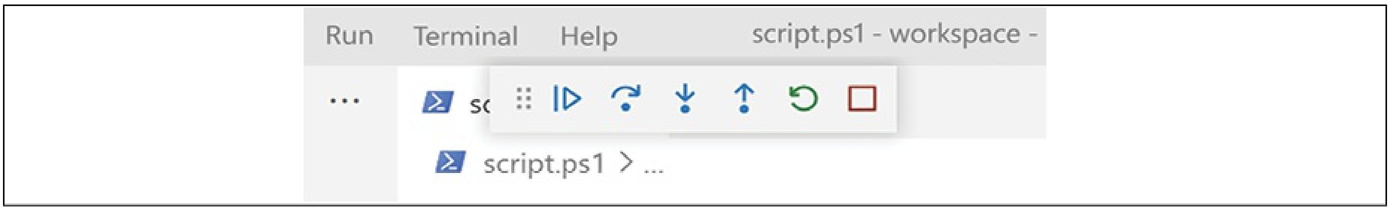


Figure 23.7: Debugger controls

Pressing the left-most icon, **Continue**, will move on to the next breakpoint.

The **VARIABLES** box on the left-hand side shows the state of each of the variables in PowerShell at the point the debugger stopped. The values of the `$name` and `$names` variables are visible and can be inspected.

Viewing the CALL STACK

The **CALL STACK** is used to record the path taken to a specific point in code. The **CALL STACK** includes a record of each command, script, or script block that was called.

A script that contains a single command with a breakpoint, such as the following script, will only contain itself in the **CALL STACK**:

```
Write-Host 'Hello world'
```

This is shown in *Figure 23.8*:

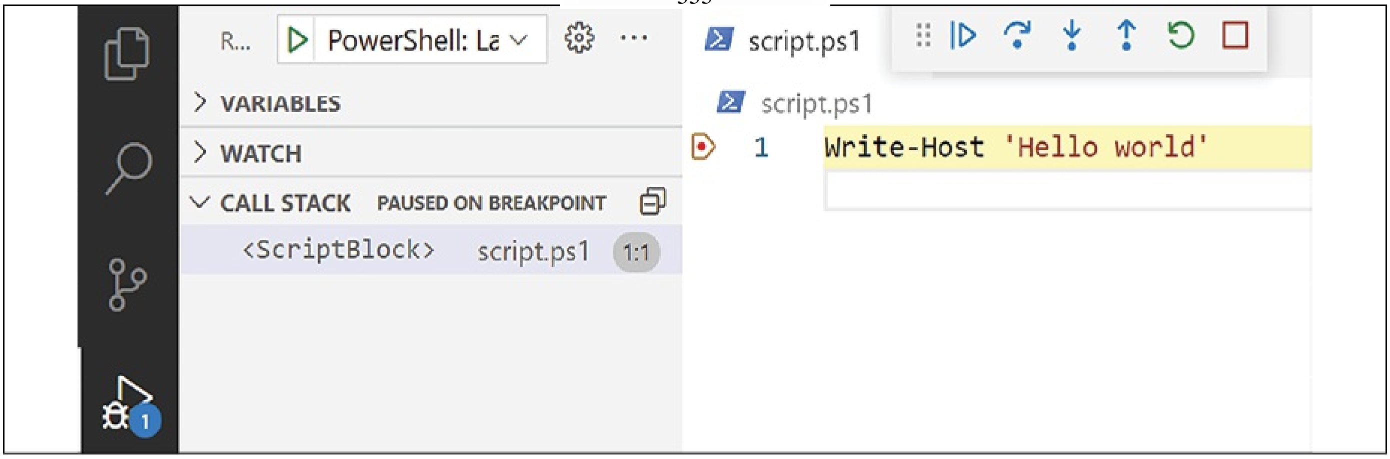


Figure 23.8: Viewing the **CALL STACK**

Each script, function, or script block that is called to get to the breakpoint is added to the **CALL STACK**. In the following script, getting to the breakpoint means calling the first function, then the second, and then the third:

```
function first {
    second
}
function second {
    third
}
function third {
    Write-Host 'Hello world'
}
first
```

Each time a function is called from inside an existing function (or script, or script block), a new value is added to the **CALL STACK**:

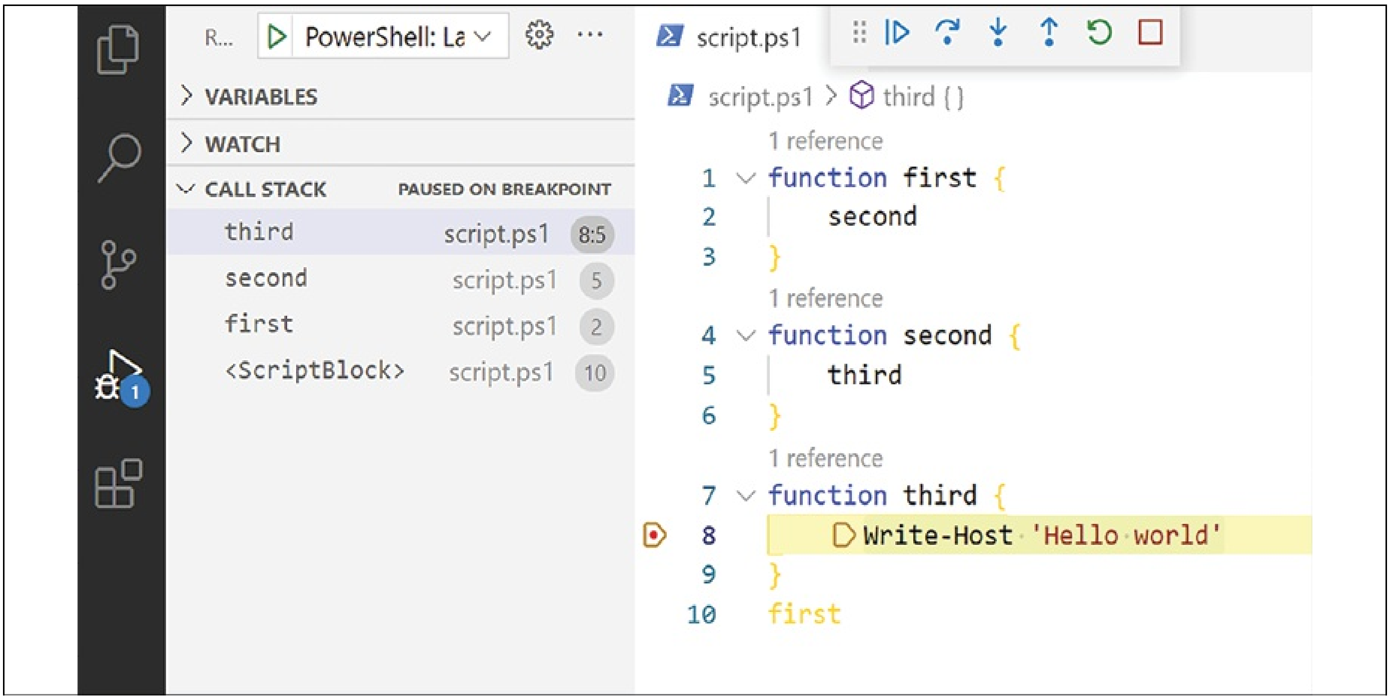


Figure 23.9: Viewing the **CALL STACK** with nested functions

Clicking on each of the entries in the **CALL STACK** will show what was called to get to the breakpoint. The final entry in the **CALL STACK** will highlight line 10.

This allows a developer to follow the path to the breakpoint even if a script is more complex.

The functions in the previous example do not make use of any variables. The **Auto** section of the **VARIABLES** window is

therefore empty.

Using launch configurations

A launch configuration is a JSON file that describes how debugging should be performed. Launch configurations are valuable if a script requires arguments. Without a launch configuration, you would have to edit a script so that it wouldn't require arguments.

Launch configurations are only available when Visual Studio Code has a directory open. If the editor was used to open a file path, the **Run and Debug** option will show the need to open a folder:

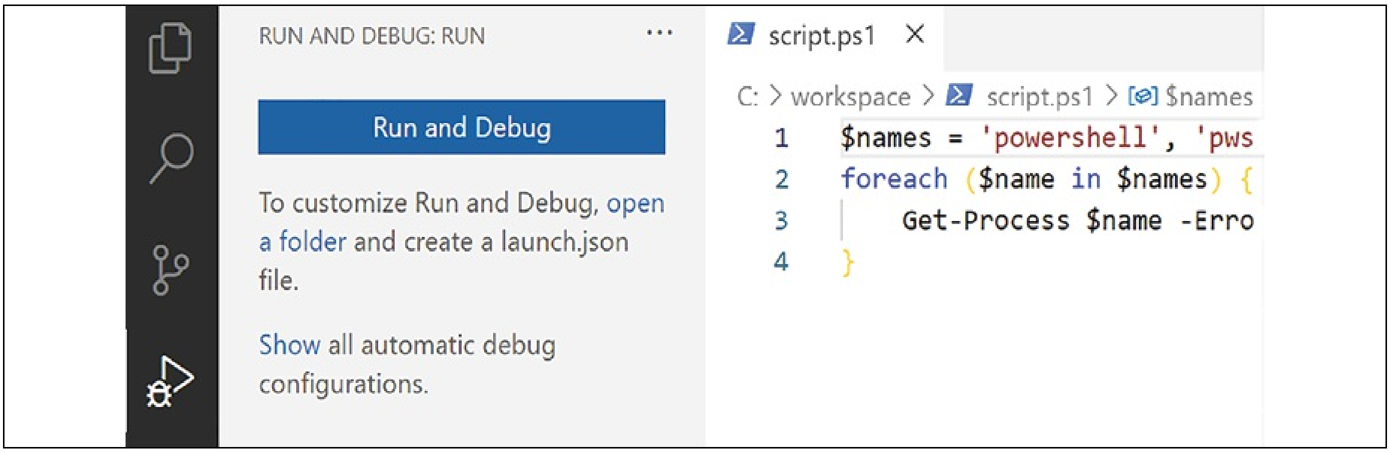


Figure 23.10: Opening a folder dialog

Selecting **open a folder** will offer a prompt for the directory containing the current script. Clicking **OK** will open that directory, and the Explorer option will contain all the files in that folder.

Returning to the **Run and Debug** option will now offer an option to create a launch configuration JSON file:

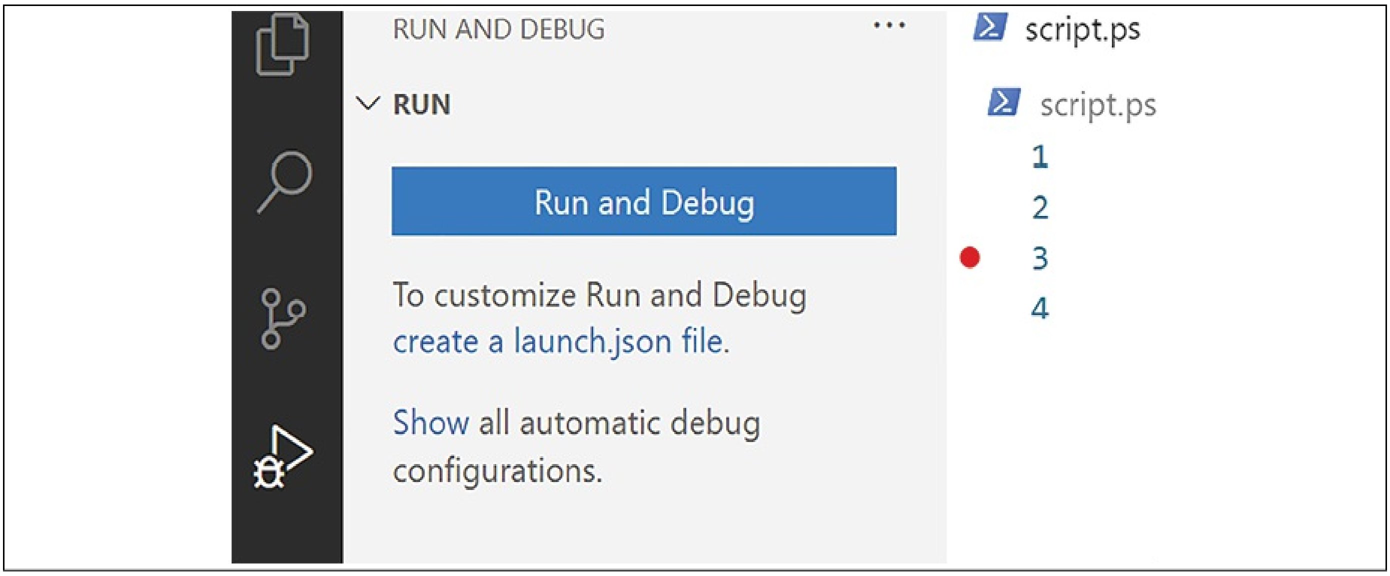


Figure 23.11: Creating a launch configuration

Clicking on the new option will open a new menu. Selecting **Launch Current File** will create a `launch.json` file in a `.vscode` folder in the current directory.

The `launch.json` file can be edited, adding a configuration that can be used to run a specific script.

Each configuration is written in JSON. In the following example, the existing entry is copied and used to create a new launch configuration. The configuration includes the `"args"` property, and a value is set to satisfy the script:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "PowerShell: Launch Current File",
            "type": "PowerShell",
            "request": "launch",
            "script": "${file}",
            "cwd": "${file}"
        },
        {
            "name": "PowerShell: Launch with arguments",
            "type": "PowerShell",
            "request": "launch",
            "script": "${file}",
            "cwd": "${file}",
            "args": [
                "-Name pwsh"
            ]
        }
    ]
}
```

A script requiring arguments can now be run in the debugger by choosing the new **PowerShell: Launch with arguments** configuration:

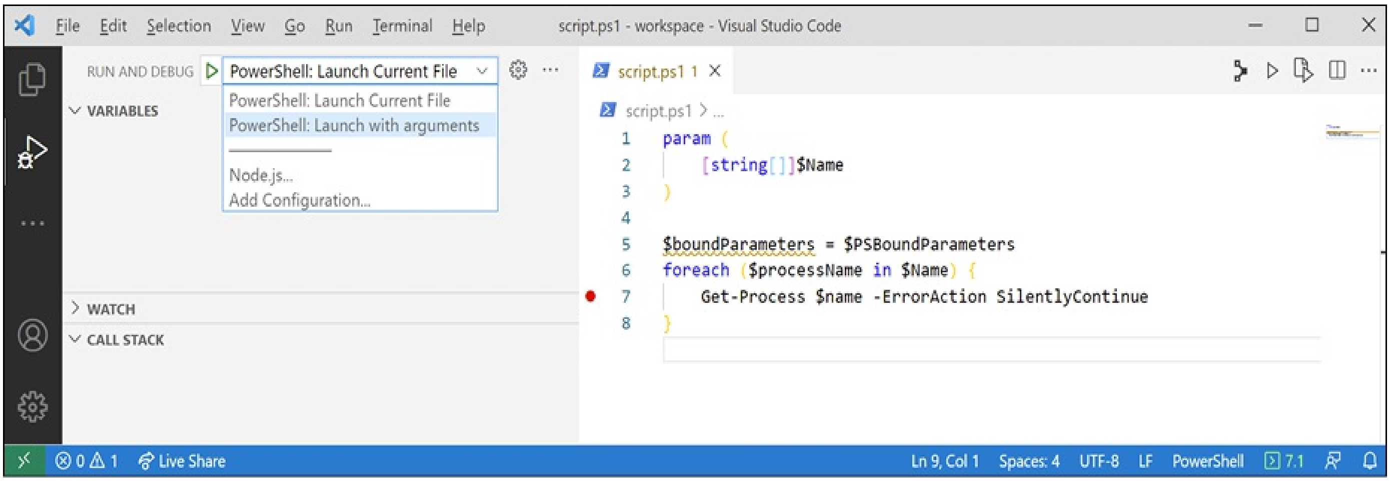


Figure 23.12: Launch with arguments

A complex script may include many variables, which can make attempting to track values using the **VARIABLES** window difficult.

Using WATCH

You can use the **WATCH** window to track a smaller number of variables or expressions as breakpoints are triggered.

The next example script sets two variables with every iteration of a loop:

```
$AValue = $ZValue = 0
for ($i = 0; $i -lt 10; $i++) {
    $AValue = $i
    $ZValue = $i * 2
}
```

The names of the variables mean they will appear at opposite ends of the **VARIABLES** window.

Each variable can be added to the **WATCH** window and the debugger will show the current value of those variables. Before the

debugger runs, the value is shown as **not available**:

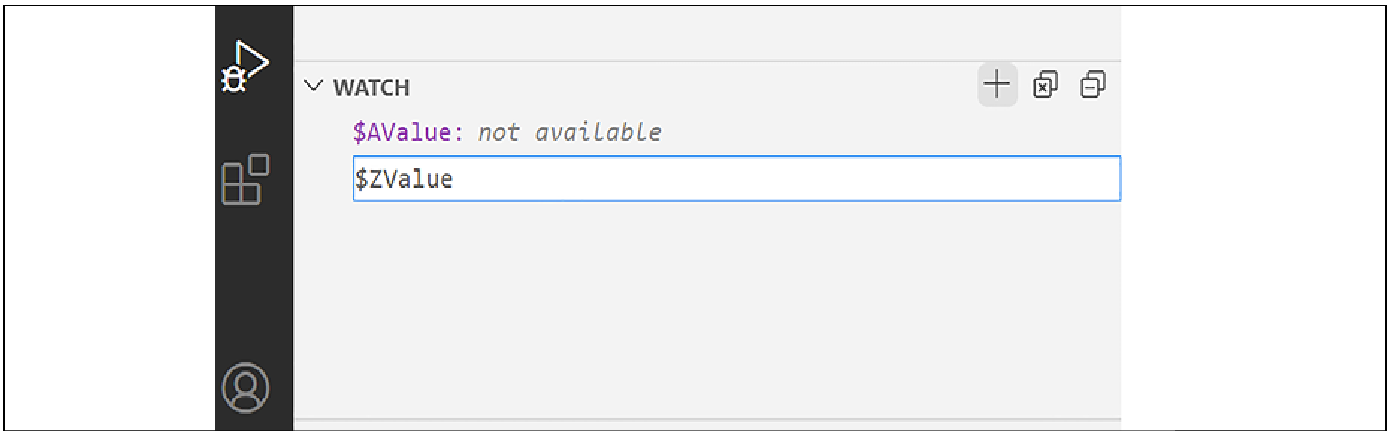


Figure 23.13: WATCH expressions

Once the debugger is started, the variables will be given their initial values as defined by the script. Then, each time **Continue** is pressed, the values will update to reflect that iteration of the loop.

For example, after the fifth press of **Continue**, the updated state of the variables is as follows:

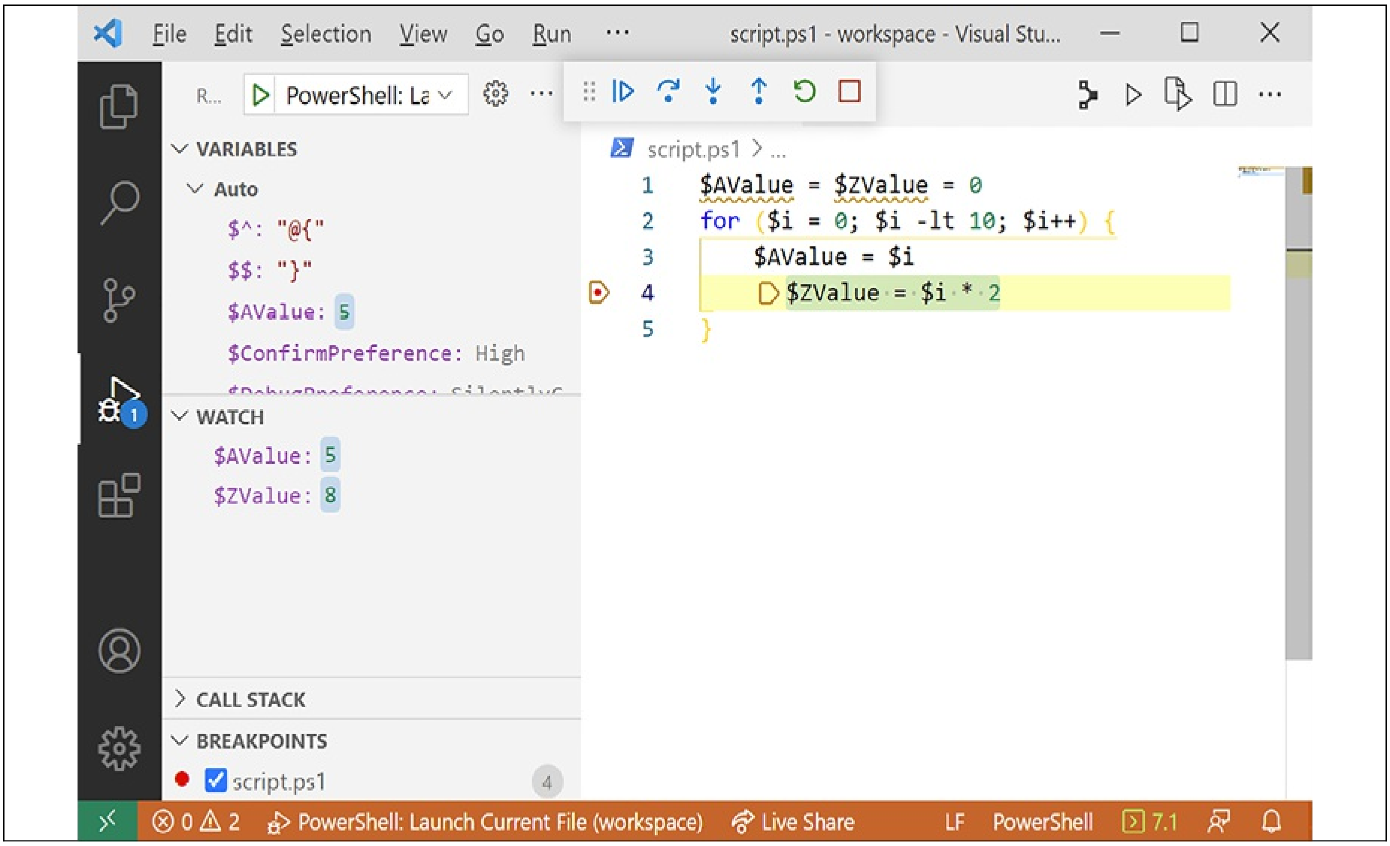


Figure 23.14: WATCH in action

The expression added to **WATCH** can be any command or statement that returns a simple value. Complex values, such as custom objects, will not display well in the **WATCH** window.

The debugging in Visual Studio Code is an especially useful tool that simplifies debugging complex scripts.

Summary

Some errors in PowerShell come up again and again, and being able to spot and identify such errors can reduce the amount of time spent attempting to isolate a problem. Several common problems were introduced based on real-world bugs.

Not every bug can be attributed to a common error, and sometimes a more extensive investigation is required. PowerShell

includes a debugger that you can use from either the command line or an editor to isolate bugs.

Visual Studio is a PowerShell editor that simplifies using the PowerShell debugger. It makes it easier for a developer to use line-based debugging.

This chapter brings this book to a close. PowerShell is full of rabbit holes to dive down and explore, more than is possible to include in this book.

Perhaps one of the most prominent features of PowerShell is consistency. Each command, each module, is presented consistently, and each has help immediately available.

The move to .NET Core and open source has opened interesting avenues for exploring the inner workings of PowerShell, fixing bugs, and extending the language.

PowerShell is supported by a fantastic friendly community, willing to help and chat regardless of experience level. User groups exist in many parts of the world, and the virtual PowerShell user group is always available: `https://poshcode.org/`

All these things combine to make PowerShell a fun language to learn and use.